Fault Interaction and its Repercussions

Nicholas DiGiuseppe
Department of Informatics
University of California, Irvine
Irvine, California 92617-3440
Email: ndigiuse@ics.uci.edu

James A. Jones
Department of Informatics
University of California, Irvine
Irvine, California 92617-3440
Email: jajones@ics.uci.edu

Abstract—**Multiple faults in a program can interact to form new behaviors in a program that would not be realized if the program were to contain the individual faults. This paper presents an in-depth study of the effects of the interaction of faults within a program. Many researchers attempt to ameliorate the effects of faulty programs. Unfortunately, such researchers are left to rely upon intuition about fault behavior due to the paucity of formalized studies of faults and their behavior. In an attempt to advance the understanding of faults and their behavior, we conducted a study of fault interaction across six subjects with more than 65,000 multiple-fault versions. The results of our study show four significant types of interaction, with one type — faults obscuring the effects of other faults — as the most prevalent type. The prevalence of obscuring faults' effects has an adverse effect on many automated software-engineering techniques, such as regression-testing, fault-localization, and fault-clustering techniques. Given that software commonly contains more than a single fault, these results have implications for developers and researchers alike by informing them of expected complications, which in many instances are opposite to intuition.**

I. Introduction

As software continues to grow more complex, the unfortunate reality is that the maintenance process grows proportionally more complex. For example, simply attempting to reduce the number of delivered faults is estimated to consume 50%–80% of the development and maintenance effort [4]. This problem becomes even more difficult as the number of faults in software increases, and real-world software almost always has multiple faults. To help alleviate this burden, researchers have developed a variety of automated techniques and methods that can assist the software-maintenance process. *Fault localization* (FL) is the process of using program spectra to find common characteristics among failing test cases in order to isolate the faulty instructions (e.g., [1], [12], [15], [17]). *Failure clustering* (FC) uses similar program spectra to try to isolate particular test cases that are failing due to the same cause (e.g., [13], [16], [20], [21]). *Regression testing* (RT) attempts to assure that an attempted fix for a fault did not in fact inadvertently cause more faults (e.g. [9], [23], [25]). Several techniques have been proposed to aid regression testing such as selection, reduction, and prioritization.

These three methods have been popular research topics for some time. Yet, despite the many gains we have seen through their results, these techniques suffer from an immature understanding of faults. All three methods utilize the same body of knowledge and attempt to leverage that information regarding fault behavior to better locate and remove faults. However, little research has been conducted to investigate the nature of faults and their interactions.

As an example consider the conferences of ASE, FSE, ICSE, ICSM, ICST and ISSTA. Between the years of 2008 and 2009 amongst all these conferences there were 27 publications about fault localization alone. Contrast this figure with the four publications — [3], [18], [29], [31] — that investigated some aspect of fault understanding. In order to achieve a more informed direction and growth in these fields, we would benefit from a better understanding of faults.

In order to gain that understanding this work examines one aspect of fault behavior: *fault interaction* (FI). FI has been observed in a variety of other works (e.g., [5], [13], [32]). We define FI as the change in the behavior of one or more faults due to multiple faults working together. This work provides a deeper investigation of the FI phenomenon than existing work. We expect that a greater understanding of FI will enable research areas that address issues with faults to better cope with the challenges that arise from the presence of multiple faults.

The main contributions of this work are

1) A more comprehensive and deeper study of fault interaction than was previously conducted. We replicate and expand the study by Debroy and Wong [5]. In contrast to the earlier study, ours utilizes real-world subjects and more faults. The use of these additional subjects reveals new findings that in some ways contradict the findings of the earlier study. In addition, we have clarified some of the definitions used in the earlier study to address some of their ambiguity and abstruseness.
2) An investigation of the factors that may be an early indication or heuristic for finding fault interaction. Because it is extremely difficult to know the number of faults that are present in a software system before finding them (which often can only take place by finding and fixing each, one-by-one), the interaction of these faults is likely to affect software-engineering techniques. If we can predict the presence of fault interaction, models may be constructed to account for these. Our investigation examined a number of factors and found one to be predictive of fault interaction.
3) A discussion and analysis of how this interaction can affect other automated software-engineering techniques

such as fault localization, failure clustering, and regression testing.

In the next section we present the background which motivates this work. In Section III, we explain the design of our experiment. Next, in Section IV, we display the results of our study. Then, in Section V, we explain the implications of those results. Next, in Section VI, we discuss some of the threats to the validity of this work. Finally, in Section VII, we provide conclusions and a discussion of future work.

II. Background

To demonstrate the implications of our study on fault interaction, we provide background on three research areas of automated software-engineering techniques: regression testing, fault localization, and failure clustering. We discuss their relation to fault behavior and interaction in Sections II-A, II-B, and II-C, respectively. We then discuss the work of Debroy and Wong [5] in Section II-D and explain the framework that it provides for our study.

A. Regression Testing

Regression Testing (RT) has been a long and widely studied field of software-engineering research. The purpose of RT is to ensure that developers do not inadvertently introduce new faults, after altering the code. RT research is often motivated by the costs of developing and maintaining test suites and the difficulty for developers to fully understand how changes may affect other parts of the program. RT research is a popular research area with a wide range of applications such as evolving configurable software [23], understanding the cost effectiveness of different RT methods [25], and selecting and prioritizing regression test cases [26], [27].

In standard regression-testing practice and in such regression-testing research, testers share an expectation of the behavior of faults — namely, that if a developer properly fixes a fault, all previously passing test cases will still pass. That is, when an existing fault is fixed and no new faults were introduced, each passing test case will continue to pass. Our investigation into fault interaction shows this assumption to be false in certain scenarios.

B. Fault Localization

Automated fault-localization (FL) techniques attempt to reduce the search space that developers need to examine when locating faults which cause failures. One way that researchers have reduced the search space is by computing slices [30], dynamic slices [14], and the use of execution slices and dices [2]. More recently, researchers have proposed statistical techniques (e.g., [1], [12], [15], [17]) to identify execution events that correlate with failure.

Intuition and experience tell us slicing and statistical FL methods are very dependent upon fault behavior. Jones et al. [12] found that the "effectiveness of the technique declines…as the number of faults increases." Multiple faults produced noise which when focused upon did not correspond to any fault, and it is difficult to differentiate the noise from the *real* fault behavior. A similar phenomenon was found by Zheng et al. [32] and Liblit et al. [15], which they refer to as *super-bug predictors*. Zheng claims that super-bug predictors "make feature selection difficult." Liblit reinforces this by claiming that super-bug predictors, "are also highly non-deterministic (because they are not specific to any single cause of failure)." Denmant et al. [6] describes coverage-based FL requiring an implicit assumption that multiple faults be independent of one another, or they will not produce "good results."

Recently, the authors of this paper found contradictory evidence suggesting that FL does not decrease in effectiveness in the presence of multiple faults [7]. We observed behavior wherein a single fault becomes easily detectable while it obfuscates or hides the remaining faults; exhibiting that fault interaction has a large impact upon FL's ability to detect faults individually and collectively.

Being able to understand how faults work together, and how fault interaction will change program behavior fundamentally affects the ability of FL techniques to locate and isolate those behavioral patterns.

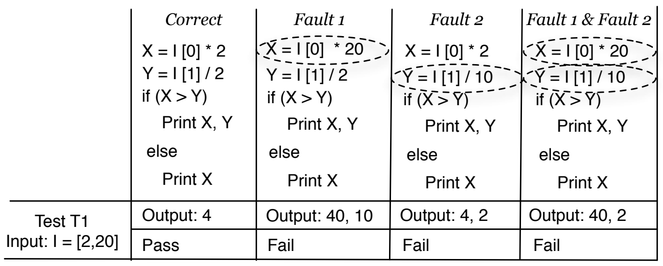
C. Fault Clustering

To separate and distinguish the effects of multiple faults, researchers have proposed fault clustering (FC) techniques. Podgurski et al. [22] group profiles of different executions in order to locate failures and predict reliability. They postulated that the unusual executions are most likely tied to failures, and can be grouped together. A number of researchers proposed other such FC techniques (e.g., [13], [16], [20], [21]).

FC faces a similar problem as FL in the presence of multiple faults. Clustering test cases based on their execution relies upon the ability to isolate particular fault's impacts upon those executions. For FC to be effective it must isolate which features of the executions have the least amount of noise or will be least affected by noise. Thus knowing which features to pick, and how to use them to be able to isolate faults becomes much more difficult. Without understanding how fault interaction will alter executions, FC techniques will likely be less than fully effective.

D. Insights from Existing Research on Fault Interaction

In 2009, Debroy and Wong [5] explored the idea of fault interference by examining the Siemens suite. In their study they examined the passing and failing statuses of test cases as different faults were added. They created 3,267 faulty versions containing anywhere from one to seventeen faults, with the highest proportion of versions lying in the range between four to thirteen faults. Their study gave evidence that when dealing with multiple faults, researchers "should not make the independence assumption" and that "interference…[was] observed approximately two-thirds of the time" [5]. Upon finding a prevalence of fault interference, they also claimed that what we will later define as *multi-type interaction* was the most prevalent of any interaction type comprising almost 50% of the interaction types observed.

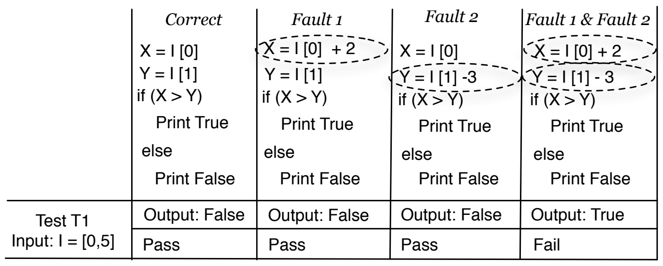


	Correct	*Fault 1*	*Fault 2*	*Fault 1 & Fault 2*

	X = I [0] * 2
	Y = I [1] / 2
	if (X > Y)
	Print X, Y
	else
	Print X
		X = I [0] * 20
		Y = I [1] / 2
		if (X > Y)
		Print X, Y
		else
		Print X
			X = I [0] * 2
			Y = I [1] / 10
			if (X > Y)
			Print X, Y
			else
			Print X
				X = I [0] * 20
				Y = I [1] / 10
				if (X > Y)
				Print X, Y
				else
				Print X

Test T1 Input: I = [2,20]	Output: 4	Output: 40, 10	Output: 4, 2	Output: 40, 2
	Pass	Fail	Fail	Fail

Fig. 1. A simple program with a correct version and faulty versions, Fault 1 and Fault 2. This example demonstrates pass/fail independence, when faults may or may not interact to change the program behavior but do not change the result of the pass/fail status.

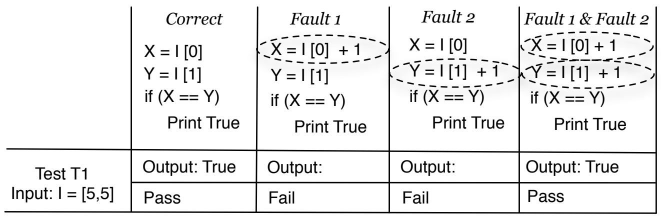


	Correct	*Fault 1*	*Fault 2*	*Fault 1 & Fault 2*

	X = I [0]
	Y = I [1]
	if (X > Y)
	Print True
	else
	Print False
		X = I [0] + 2
		Y = I [1]
		if (X > Y)
		Print True
		else
		Print False
			X = I [0]
			Y = I [1] -3
			if (X > Y)
			Print True
			else
			Print False
				X = I [0] + 2
				Y = I [1] - 3
				if (X > Y)
				Print True
				else
				Print False

Test T1 Input: I = [0,5]	Output: False	Output: False	Output: False	Output: True
	Pass	Pass	Pass	Fail

Fig. 2. A simple program with a correct version and faulty versions, Fault 1 and Fault 2. This example demonstrates fault synergy, when multiple faults combine to cause a failure none could create on their own.



	Correct	*Fault 1*	*Fault 2*	*Fault 1 & Fault 2*

	X = I [0]
	Y = I [1]
	if (X == Y)
	Print True
		X = I [0] + 1
		Y = I [1]
		if (X == Y)
		Print True
			X = I [0]
			Y = I [1] + 1
			if (X == Y)
			Print True
				X = I [0] + 1
				Y = I [1] + 1
				if (X == Y)
				Print True

Test T1 Input: I = [5,5]	Output: True	Output:	Output:	Output: True
	Pass	Fail	Fail	Pass

Fig. 3. A simple program with a correct version and faulty versions, Fault 1 and Fault 2. This example demonstrates fault obfuscation, when faults interact to hide each other changing a failing test case into a passing test case.

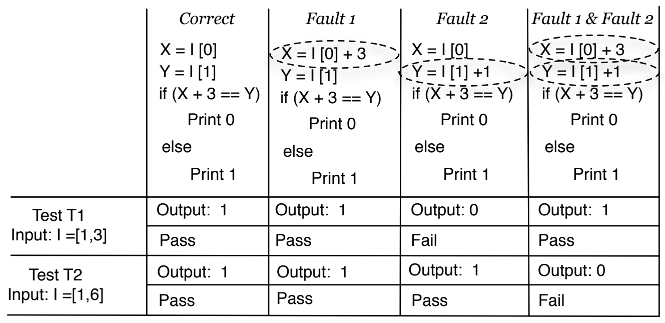
In their examination they classified four different types of *fault interference* which were named based upon wave-theory principles. However, we found that these titles lead to a lack of precision when talking about these sets due to their implications. In order to facilitate elucidative discourse, we have renamed their categories in such a way that we believe will enable greater precision and less confusion. For example, "interference" is most commonly used outside of wave theory to mean partial or complete obstruction. Therefore calling it fault interference implies that the faults are blocking each other in some way, which is not always the case. Thus, in this paper we use *fault interaction* (which we are abbreviating as "FI"). Another example is Debroy and Wong naming a particular type of interaction as fault independence. The term "independence" implies that the faults in no way affect each other. However because the data informing the determination of independence is gained at a test-case pass/fail-status level, it is possible for two faults to interact, and yet still be classified into this *independence* category. Looking at Figure 1 you can see an example of independence, despite the fact that faults are interacting. This figure illustrates two separate faults: Fault 1 and Fault 2, each causing a program failure along with how their output differs from the correct version. When those two faults are combined there is still a failure, even though those faults interacted in a way to create output which was previously unknown. This combination would have been classified as independent, which while it conforms to their definition of fault independence, clearly demonstrates an interaction between the two faults. The example shows the faults are interacting in a way to change program behavior and thus are not strictly independent, but they are independent with respect to the pass/fail statuses of the test cases.

The previous example demonstrates the FI type we classify as *pass/fail independence*. Pass/fail independence occurs when multiple faults either do not interact, or the fault interaction occurs in such a way as to not change the pass/fail status of the program. Next, we demonstrate the remaining types of FI that may occur in a program.

Figure 2 shows two faults interacting in a way that causes the program to fail, even though neither individual fault causes failure when present in the program by itself. This figure shows that Faults 1 and 2 pass test case T1, however, when both are present simultaneously, their interaction causes a new failure that neither could cause alone. This is classified as *fault synergy* because the faults are working together to produce a new failure.

Figure 3 demonstrates an opposite phenomenon to fault synergy: two faults which individually cause failure but together cause a passing behavior. This situation is classified as *fault obfuscation* because the faults interact in a way that they become hidden from the developer because they are no longer causing failures. It is significant to note in this example that both faults obfuscated each other. For fault obfuscation to occur, only a single fault needs to have its failure turn into a pass, effectively hiding it from the developer.

The final remaining type of FI is demonstrated in Figure 4. What we see in this figure is that when Fault 1 and Fault 2 are combined, they work together to cause multiple types of FI. When we look just at Test T1 we see that Fault 2 is obfuscated because its failure becomes hidden. We have already described this type of interaction as fault obfuscation. When we look just at Test T2 we see that the faults work together to change a pass into a failure. We have already described this type of interaction as fault synergy. Because this example shows both synergy and obfuscation within the same test suite, we classify this as the final new type, *multi-type* interaction. Multi-type interaction only occurs when both obfuscation and synergy are present within the same test suite. However it should be noted that they need not be in equal proportions. If a single test case experiences obfuscation and in the entire rest of the suite the remaining test cases experience synergy, we would still classify it as a multi-type.



	Correct	*Fault 1*	*Fault 2*	*Fault 1 & Fault 2*

	X = I [0]
	Y = I [1]
	if (X + 3 == Y)
	Print 0
	else
	Print 1
		X = I [0] + 3
		Y = I [1]
		if (X + 3 == Y)
		Print 0
		else
		Print 1
			X = I [0]
			Y = I [1] +1
			if (X + 3 == Y)
			Print 0
			else
			Print 1
				X = I [0] + 3
				Y = I [1] +1
				if (X + 3 == Y)
				Print 0
				else
				Print 1

Test T1 Input: I =[1,3]	Output: 1	Output: 1	Output: 0	Output: 1
	Pass	Pass	Fail	Pass
Test T2 Input: I =[1,6]	Output: 1	Output: 1	Output: 1	Output: 0
	Pass	Pass	Pass	Fail

Fig. 4. A simple program with a correct version and faulty versions, Fault 1 and Fault 2. This example demonstrates multi-type interaction, when faults interact to cause obfuscation and synergy within the same test suite.

We utilize this framework of fault interaction provided by Debroy and Wong. However, in addition to our renamed interaction types, we replicate their study while expanding theirs in a number of ways. We provide an analysis of the effects of FI upon a variety of popular research practices (FL, FC, and RT) along with the factors which influence FI including: size of program, number and percentage of data and control dependencies, number and percentage of branches, and number and percentage of methods. Further, in our choice of subjects, we include multiple real-world subjects, the details of which are discussed in III-A, whereas Debroy and Wong utilized only the Siemens suite which consists of seven very small ($<$ 600 LOC) programs. Lastly, we have over 20 times more faulty versions, and our versions have a more even distribution among the different quantities of faults.

In addition, Jia and Harman [10] researched the effects of a similar concept to fault interaction in the form of *higher-order mutation*. Similar in concept to a standard software mutant, which contains an artificial fault, a higher-order mutant is the combination of multiple artificial faults such that their effect on a program is more than the sum of their individual parts. Jia and Harman speculate that higher-order mutants are more similar to real faults than standard mutants. Santelices et al. [28] provided a formal model to predict when multiple changes to evolving software made by multiple developers produce run-time interaction. In contrast, our goal is to investigate the prevalence and characteristics of fault interaction in the domain of software testing.

III. Experiment Design

In order to understand how faults interact with each other across real-world programs and how that interaction affects FL, FC, RT, we conducted the following experiment. In this section we first discuss the subjects that we used in our experiment. Next, we discuss the independent and dependent variables that we utilized, and finally we discuss the setup of our experiment.

A. Subjects

In this experiment we used six C-language subjects: Flex version 2.5.4, Gzip version 1.0.7, Replace, Schedule, Sed version 3.02, and Space. Out of our six subjects, four are real world programs: Flex, Gzip, and Sed, which are Unix utility programs and Space, which is an interpreter for an array definition language. The other two, Replace and Schedule are from the so-called “Siemens suite.” These programs vary substantially in size as follows: Flex has 14273 LOC, Gzip has 7928 LOC, Replace has 563 LOC, Schedule has 509 LOC, Sed has 10154 LOC, and Space has 6445 LOC.

All of these programs were obtained from the Subject Infrastructure Repository (SIR) [8] along with their faulty versions and test cases. In subjects where the number of faulty versions presented by SIR was less than 20, we added additional faults through random mutation. We implemented the methods described by Offutt et al. [19] in which one randomly selects a line to mutate, and then randomly selects an applicable mutation operator in order to create mutants which form a representative set.

As with previous researchers (e.g., [11], [12], [17], [32]), we excluded those faulty versions which caused no failures — in the case that this exclusion reduced the number of faulty versions below 20, random mutation was performed to replace that fault, and ensured that no fault lines overlapped, where a fault is between 1 and n lines of code.

B. Variables and Measures

Because our primary goal was to understand how FI affected program behavior, our experiment varied a single independent variable — the number of faults in a program — and examined four dependent variables: (1) *fault synergy*: the number of test suites where the faults interacted to create new failures, (2) *fault obfuscation*: the number of test suites where the faults interacted to create less failures, (3) *pass/fail independence*: the number of test suites where the faults’ interaction did not change the pass/fail status of the program, and (4) *multi-type*: the number of test suites where the faults interacted to cause synergy and obfuscation simultaneously. To enable a better understanding of each of these dependent variables, we will give their definition in set notation.

For n faults in program ρ, we denote each fault by $f_1, f_2, \ldots, f_n$. Let ρ_i denote a variant of program ρ which contains fault f_i where ($1 \leq i \leq n$). Therefore, each program ρ_i represents a program with a single fault. We then say that t represents a test suite for any ρ_i. Executing all tests in t on a corresponding program ρ_i will return a set of failing test cases which are denoted v_i. We will then say that the set S is composed of all the unions of each of the failing test cases v_i for programs $\rho_1, \ldots, \rho_n$ such that: $S = v_1 \cup v_2 \cup \ldots v_n$. We then let ρ^M denote the program ρ which contains all the faults $f_1, \ldots f_n$ which are contained in programs $\rho_1, \ldots, \rho_n$. Executing the same test suite t on program ρ^M yields a resulting set M which represents the failing test cases from program ρ^M.

TABLE I

THE NUMBER OF VERSIONS, SIZE OF TEST SUITE, AND NUMBER OF EXECUTIONS FOR EACH OF OUR SIX SUBJECT PROGRAMS

	Flex	Gzip	Replace	Schedule	Sed	Space	**Total**
1 Fault	21	20	25	20	20	33	139
2 Faults	189	182	185	165	188	200	1109
3 Faults	502	505	598	500	660	200	2965
4 Faults	1000	1000	700	500	950	200	4350
5 Faults	1000	1000	700	500	950	200	4350
6 Faults	1000	1000	700	500	950	200	4350
7 Faults	1000	1000	700	500	950	200	4350
8 Faults	1000	1000	700	500	950	200	4350
9 Faults	1000	1000	700	500	950	200	4350
10 Faults	1000	1000	700	500	950	200	4350
11 Faults	1000	1000	700	500	950	200	4350
12 Faults	1000	1000	700	500	950	200	4350
13 Faults	1000	1000	700	500	950	200	4350
14 Faults	1000	1000	700	500	950	200	4350
15 Faults	1000	1000	700	500	950	200	4350
16 Faults	1000	1000	700	500	950	200	4350
17 Faults	1000	1000	700	500	950	200	4350
Total Versions	14,712	14,707	10,608	7,685	14,168	3,233	65,113
Size of Test Suite	527	214	5,542	2,560	363	13,527	22,733
Number of Executions	7,753,224	3,147,298	58,789,536	17,673,600	6,253,984	43,732,791	138,239,433

These sets S (the set of failures for the union of single-fault versions) and M (the set of failures for the multi-fault version) can be compared to see which of our FI types they align with:

$S \subset M$ means fault synergy has occurred because there are new failures when the faults are combined into any single program. In other words, each test case that failed on a single-fault version also failed in the multi-fault version, and also new failures occurred in the multi-fault version which did not in any single-fault version.

$S \supset M$ means fault obfuscation has occurred because there are fewer failures when the faults are combined into the multi-fault version. In other words, some test cases that failed on some single-fault version do not fail on the multi-fault version — they are effectively hidden.

$S = M$ means pass/fail independence has occurred because every test case that failed on the any single-fault version also failed on the multi-fault version and any test case that failed on the multi-fault version also failed on one of the constituent single-fault versions.

$(S \not\subset M) \wedge (S \not\supset M) \wedge (S \neq M)$ means multi-type interaction has occurred. In other words, there exists at least one test case that fails on some single-fault version which does not fail on the multi-fault version and also that at least one test case fails on the multi-fault version which does not fail on any single-fault version.

C. Experiment Setup

In this experiment we gathered the pass/fail status for our six subject programs containing between 1 and 17 faults. For each subject, we compared the pass/fail information of a multi-fault version with the results from the union of the pass/fail status of the single-fault versions which composed the multi-fault version (see Section III-B). Utilizing that comparison, we found which type of FI that version contained.

We determined a program's pass or fail status by comparing its output to an oracle version, which contained none of the faults that we were enabling or disabling. When there was any difference in the output between the faulty version and the oracle version, the test was marked as a failure. If the outputs were equal, it was considered a pass.

In order to more clearly understand the different types of FI and how they affected our subjects we created faulty versions of our programs containing 1, 2, …, 17 faults, which were chosen at random. This process consisted of selecting random faults, injecting those faults in the program, executing the program, capturing the output, and determining the pass/fail statuses of each test case. This process was repeated for a set time period which was roughly equal for each subject. Thus, a program with a small test suite has more multi-fault versions because it required less time to run. However, if a subject had a large test suite, we created fewer faulty versions because executing the entire test suite required more time. Table I lists the number of versions that we generated and experimented with for each subject at each quantity of faults. This table shows the number of faults on the left, how many versions we created for that quantity of faults for each of the six subjects, and the total number of versions created for that quantity of faults on the right. At the bottom some statistical information is given regarding the total number of faulty versions made for a particular subject, that subject's test-suite size, and the total number of executions which were run with that subject along with the total amount when considering all six subjects. As can be seen, over 65,000 faulty versions were generated and 138,239,433 test cases were executed to gather our data. After all the data was gathered for all the subjects, we then performed our comparisons as described in Section III-B.

IV. RESULTS

We present our results relating FI occurrence and frequency in Figures 5, 6, 7, 8, 9, and 10. In each figure the horizontal axis represents our independent variable: the quantity of faults that existed in a particular version. Along the vertical axis we represent the frequency of a FI type as a percentage. At each coordinate along the horizontal axis there are four bars, each representing a single type of FI that can exist in a program: fault obfuscation, fault synergy, pass/fail independence, and multi-type interaction. Each bar is comprised of all the faulty versions which were generated for that quantity of faults. More simply, at a quantity of faults n, the bars represent the different FI types for every n faulty version that exists for that program. In other words, the bars are not classified by individual faults, but instead by the quantity of faults that comprised all such versions. Also note that in many places there are less than four bars because at some points not all four types of FI were observed. Also at quantity of faults 1, there are no bars for any program because FI cannot occur if there is only one fault.

Figures 5, 6, 7, 8, 9, and 10 represent the FI results for the subjects Flex, Gzip, Replace, Schedule, Sed, and Space, respectively. Every subject except Gzip follows a similar pattern of a monotonically increasing fault-obfuscation value. Out of those five subjects all except Space experience fault obfuscation becoming the most prominent FI type constituting more than 70% of the FI at the 17 fault level.

In Figure 11 we display the relationship between the quantity of faults and the percentage of test-suite failure. Along the horizontal axis we again represent the quantity of faults such that as you move right along that axis the quantity of faults will increase. On the vertical axis, we represent the percentage of test-case failure for a given subject. There are seven different lines: a single line for each subject and also a “total” line. The total plot line represents the aggregation of all our six subjects. Also, note that the plot lines between fault quantities are drawn only to assist the reader in tracking the trends in the plot — the intermediate plot points (for example, between fault quantity 1 and 2) do not represent real values. Also note that for all lines when there are 0 faults, there is 0% failures — this is because a fault-free program cannot fail. However, what is notable here is that every subject’s test-suite failure value monotonically increases as you move right along the horizontal axis, despite the fault obfuscation that was found in our results. Also, every subject’s test-suite failure value is quite high, above 80% in every case, in the versions which contain 17 faults.

V. Analysis and Discussion

In order to better understand the results from Section IV, this section will provide an analysis and interpretation of them. We will begin by analyzing what types of fault interaction is found in our programs, and identify factors in those programs which influenced the causation of those interactions. We then will discuss and analyze the ramifications those interaction types have on FC, FL, and RT.

A. Fault Interaction

To identify the prevalence of fault interaction, we examine Figures 5, 6, 7, 8, 9, and 10 which display the percentage of each FI type for each subject. What is first apparent is that in five of the six subjects fault obfuscation is very prominent. We see that as the number of faults increases so does the percentage of fault obfuscation. In four of those subjects we see over 70% obfuscation at the seventeen fault level. In these subjects as fault obfuscation increases, pass/fail independence (the second most prominent type for most subjects) generally decreases along with other types of fault interaction. This means that as the number of faults increases, the subjects actually fail less often than would be expected given each individual fault’s pass/fail status. In other words, as the number of faults increases, it is more common that a single fault, or subset of faults, obfuscate the remaining faults causing the program to not fail as often. Upon further investigation into this phenomena, we found that certain faults were not being executed or were having their impact minimized.

However, when examining Figure 11, we can see the percentage of test-suite failure for each of the subjects as the number of faults increases. We see the number of faults is proportional to test-suite failure for every subject. We also see that around six faults there is a plateau. Eventually four of our subjects fail around 80% of their test suite and the other two fail around 95% of their test suite. This follows our intuition that more faults in the code brings more undesired outputs.

The harmony between these two sets of results — (1) that most often faults obfuscate each other and hide their effects so there are less failures, and (2) as the number of faults increases so too do the number of failures — lies in the subtlety of their relationship. Generally it is true that when there are more faults there are more failures. However, there aren’t as many failures as would be expected through the composition of the single faults. This obfuscation occurs because as the number of faults increases, the likelihood that a single fault or subset of faults obfuscates the remaining faults increases, causing test cases which would normally fail, to pass. Yet not all of a fault’s effects are obfuscated, only a portion of test cases which otherwise fail are caused to pass because of this obfuscation. The remaining un-obfuscated test cases still cause their failures which explains the increases in the overall percentage of test-suite failure as the quantity of faults increases. More faults do cause more failures, just not as many as would be expected.

There is a distinct difference in the results of three of our subjects. The two Siemens suite programs, experience a substantial amount of multi-type interaction. When also considering Gzip, those three subjects are the only subjects to display any significant quantity of fault synergy. Finally, Gzip’s interaction types are very unique when compared to the remaining subjects.

B. The Factors Which Affect Fault Interaction

To identify the cause of the Siemens suite programs behaving differently than the real-world subjects, and to identify why Gzip acted so uniquely, we performed a small secondary experiment to examine the factors in each program which impact FI. The factors we examined are: the lines of executable code (LoEC), the number of branches, the ratio of branches to LoEC, the number of methods, the ratio of methods to LoEC, the number of data dependencies, the ratio of data

⬛ Fault Synergy 🟦 Fault Obfuscation 🟥 Pass/Fail Independence ▨ Multi-Type Interaction

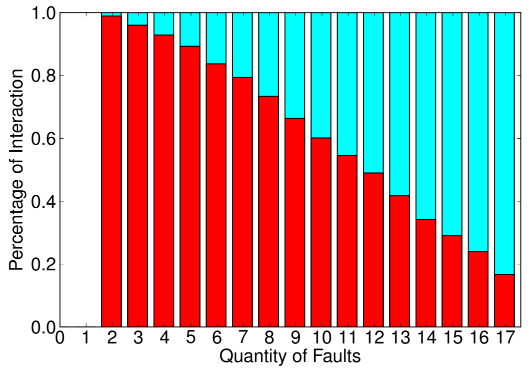


Fig. 5. The fault interaction data for the subject Flex. As the number of faults increases the amount of fault obfuscation increases dramatically and monotonically.

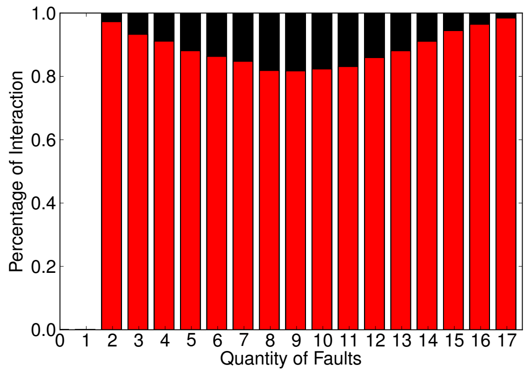


Fig. 6. The fault interaction data for the subject Gzip. This subject has a high percentage of pass/fail independence with a small presence of fault synergy which peaks at 10 faults.

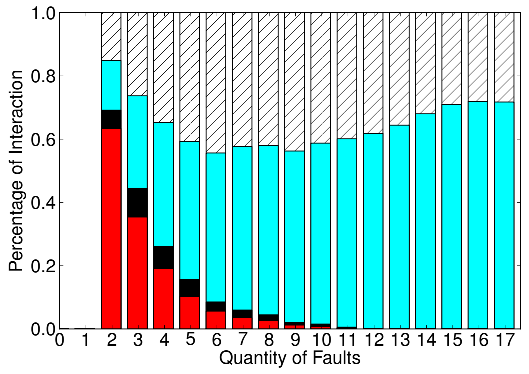


Fig. 7. The fault interaction data for the subject Replace. As the number of faults increases the amount of fault obfuscation increases dramatically and monotonically.

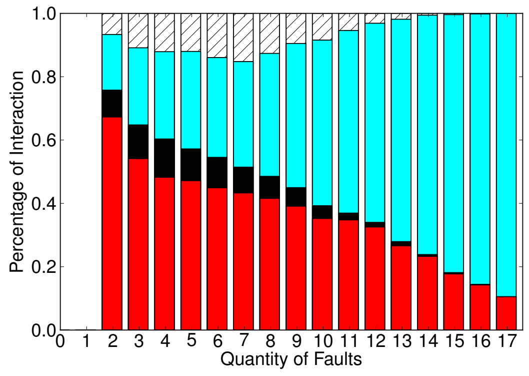


Fig. 8. The fault interaction data for the subject Schedule. As the number of faults increases the amount of fault obfuscation increases dramatically and monotonically.

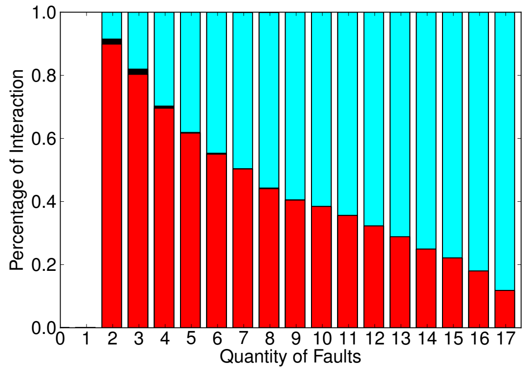


Fig. 9. The fault interaction data for the subject Sed. As the number of faults increases, the amount of fault obfuscation increases dramatically and monotonically.

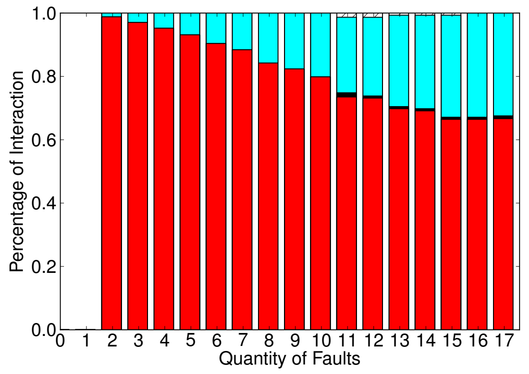


Fig. 10. The fault interaction data for the subject Space. This subject has a large subset of fault obfuscation which increases proportionately with the amount of faults.

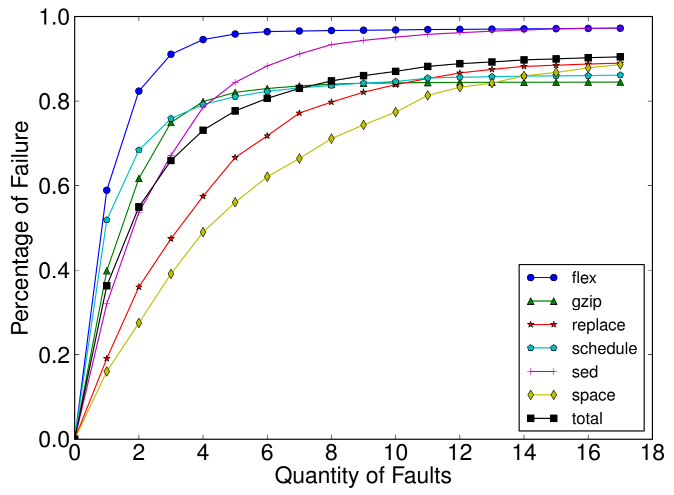


Fig. 11. The data relating how the quantity of faults corresponds with the percentage of test-case failure. Almost all subjects fail 80% of their test suites with less than 6 faults in the program, but at this point their amount of failure increases slowly.

TABLE II
THE FACTORS WHICH WE ANALYZED TO DETERMINE THEIR IMPACT ON FI.

	Gzip	Flex	Replace	Schedule	Sed	Space
LoEC	2013	4020	242	150	2253	3651
Branches	1727	2680	182	66	2554	1190
Methods	514	1261	63	41	416	717
Data Dep	45291	143075	2005	1327	94614	29883
Control Dep	26766	61279	1444	1074	21991	25047
Total Dep	72057	204354	3449	2401	116605	54930
LoEC : Branches	1.165	1.5	3.818	2.273	1.893	3.068
LoEC : Methods	3.916	3.188	3.841	3.659	5.416	5.092
LoEC : Data Dep	0.044	0.028	0.121	0.113	0.024	0.122
LoEC : Control Dep	0.075	0.066	0.168	0.140	0.103	0.146
LoEC : Total Dep	0.028	0.020	0.070	0.062	0.019	0.067

dependencies to LoEC, the number of control dependencies, the ratio of control dependencies to LoEC, the total number of dependencies, and finally the ratio of total dependencies to LoEC, as can be seen in Table II. We found that in harmony with Debroy and Wong's intuition [5], but in contradiction to their results, the LoEC a program has has a direct relationship to the type of interaction faults experience. Comparing the relationships with our FI data we find the only significant relationship is lines of executable code. The others were not significant to explain any program behavior. The two Siemens suite programs each have less than 250 LoEC and both were the only subjects to experience multi-type interaction. Further, we see that smaller programs tended to have less pass/fail independence. This is most likely due to a saturation of faults. If each fault is a single line of code, when at seventeen faults, more than 10% of Schedule's executable code is faulty. We find that because all of Debroy and Wong's results are based upon different Siemens suite programs, their results on this subject do not generalize well because Siemens programs are not large enough to demonstrate real-world behavior. However, for all the remaining factors, we found no substantial impact upon the programs, and program size does not explain Gzip's behavior.

Gzip consistently had a high proportion of pass/fail independence and showed the most substantial amount of fault synergy. We investigated this aberration and found a few characteristics of the subject program that may account for the difference in fault-interaction behavior. In Gzip, almost every test case, of a certain type, executes almost every function. Specifically, the test cases in Gzip fall into two main categories: those that cause Gzip to compress, and those that cause Gzip to decompress. The coverage between these two sets is largely different, but the coverage within each category is highly similar. Thus, faults causing failures, also tend to fit within one of these two categories, breaking compression or breaking decompression. Moreover, all other faults in that same category are almost guaranteed to be executed as well. So, when one fault is executed, all others are (of the same type) as well. This fact is also coupled with another characteristic that we found in Gzip: it is highly sensitive to fault infection. That is, when a fault is executed, a failure is almost assured. Thus, given that the existing faults (of each type) are all executed, and almost all of them cause failures, it is unlikely for them to obfuscate each other (and cause less failures). In the rare case where multiple individual single faults do not cause failures, the presence of other faults may cause synergy (more failures).

C. *Ramifications for Research and Practice*

The prevalence of fault obfuscation — resulting in fewer failures and hidden faults — has a likely substantial impact on many tasks in both research and practice. We examine three popular tasks — fault localization (FL), failure clustering (FC), and regression testing (RT) — that are popular topics for research given their costs in practice.

For FL, a developer or an automated technique attempting to find a fault causing a failure will likely face difficulties if the fault sought is not the most dominant, or obfuscating, fault. The faults that are being obfuscated may still cause failures, but at least some of the test cases that would have failed in its presence alone do not fail when another fault is also present. Such situations can cause statistical FL techniques to consider the faulty instructions of the obfuscated fault to be less correlated with failure, because they demonstrated that they can be executed while the test case passes. In such cases, the statistical analysis is "deceived" to consider those instruction to be less "suspicious" because their execution can produce passing test cases. Moreover, techniques such as dicing techniques (e.g., [2], [24]) may remove those faulty instructions from the result set altogether.

To cluster failures in order to determine the number of faults causing failures, and to determine which faults are causing which failures, FC faces a number of challenges in the face of fault interaction. Much like in the case of FL, effective FC can be undermined by the presence of instructions that are faulty, but in some cases do not cause failures because of other obfuscating faults. FC often attempts to identify "features" (often instruction execution) that distinguish certain behaviors from others. When those instructions do not cause failures in some cases, but do in others, they can be viewed as more "noisy." Thus FC suffers from fault obfuscation in that some of the potentially most informative features — namely the fault causing failures — may be utilized to a lesser degree. Also, much of FC research attempts to determine the ideal point which to stop clustering — in other words, "how many clusters should be generated?" Ideally, the number of clusters would be equal to the number of faults in the system, and the test cases in each cluster would include only those that were caused by those failures. However, given the prevalence of fault interaction seen in our results, the determination of this stopping-point of clustering and the membership of the clusters is complicated by this interaction. Moreover, because faults are being obfuscated, which results in hidden failures, an iterative application of FC is likely — clustering some failures, fixing them, and then repeating the process until failure-free. The number of iterations of this process can be difficult to determine in the presence of such fault interaction.

This same difficulty of determining the number of iterative fault finding and fixing can cause complications with time estimation in regression testing. Because each time a fault is fixed there is a high chance that a previously obfuscated fault will become active, test cases which were previously passing can and do become failing. This leads to the complication where a developer will make a change to fix a fault, and while that test case's result changes from a failure to pass, a different test case will go from passing to failing because a previously obfuscated fault is now active. We can see in Figures 5, 7, 8, 9, 10 that because of the prevalence of fault obfuscation, this kind of behavior will unfortunately be quite common, and will be especially common in programs with many faults. The case where a passing test case becomes failing may cause a developer to question whether they have introduced a new fault. As a result, developers may require extra time to investigate their change to ensure that no new faults were introduced, which can add costs to the project. Understanding such fault interactions can inform researchers and practitioners of such situations, which may lead to new cost estimation models for regression testing and debugging.

VI. Threats to Validity

Some difficulties in constructing external validity in this work deal with the ability to generalize across different software systems. We evaluated only six programs, and because of this we cannot for certain make claims that our results will hold true for any piece of software. However, because four of our subjects are real-world software, we believe this increases the chances for generalizability across all software. An additional threat to the generalizability of our study is that all of our subjects are "filter"-type programs — they take in a single input and output a single output. While other software may have more independence between modules, within each of those modules is likely to exhibit much the same interactions and issues found here.

Another issue that we faced was the need to add mutants into our subjects. The nature of this study required a certain quantity of faults, and the SIR [8] did not provide a sufficient number for some of the subjects. Because by definition mutants are not real faults, we cannot generalize their results with absolute certainty. However, Offutt et al. [19] describe a methodology which allows for the creation of mutants which are representative of real faults. To minimize this issue, our study followed the methods of Offutt et al. in all our mutant generation. In addition, we believe mutation to be an appropriate mechanism to study fault interaction because faults are simply comprised of differences in code from a theoretical correct program, much like mutants. Real faults may have more complex differences than mutants, however, these would likely serve to demonstrate even greater interactions than simpler mutants. Moreover, we did study real faults in some of our subjects and found no appreciable difference in our results from the mutantation faults.

A difficulty in creating construct validity is in how we measure FI. We measure how faults react at the test-case level, and specifically in how they affect the pass/fail status of a program. Because of this it is possible that we will entirely miss certain types of interactions like those described in Figure 1. While it is true that we could be missing many different instances of FI, the domain of our study is limited to those faults which do affect the pass/fail status of a program, and our results show that this type of interaction is both prominent, and prevalent among all our subjects.

VII. Conclusions and Future Work

In this paper we have presented an experiment to reveal the nature of FI in programs with different quantities of faults. We provided evidence which demonstrated that fault obfuscation was the most prominent of all FI types. This means that faults will often hide the impact of other faults leading to less failures than would normally be expected. We also presented data demonstrating that the number of faults is proportional to the amount of fault obfuscation most programs have. We further provided an analysis of how fault obfuscation can affect the prominent debugging fields of FL, FC, and RT.

We found that FI — particularly fault obfuscation — was a real concern for research and practice. Obfuscation will provide difficulty when attempting to perform time estimation, and when attempting to isolate multiple faults simultaneously. This also provides difficulty in RT when after fixing faults, test cases change from passing to failing because a previously hidden fault is now affecting the system.

We also provided a deeper investigation than previous work into the nature of these areas. We created more than 65,000 multi-fault versions along with providing a clarification and simplification of terms which was needed due to potential confusion in the previously used vocabulary. Our work is somewhat based upon Debroy and Wong — our studies have benefitted from their framework that they set. However, our findings are in contradiction to their work in both the prominence of different types of FI and the factors which lead to FI. Through an in-depth investigation of the factors which affect FI among different programs, we found that this can be primarily explained due to the size of our programs. Our data displayed that smaller programs, will have less pass/fail independence, and will also contain multi-type interaction which is almost nonexistent in our real-world subjects.

While our results generate valuable information regarding how faults interact with each other in real-world software and how that interaction affects popular software-maintenance research and practice, more experimentation needs to be conducted to confirm that these results generalize. Specifically, we are planning experiments on how faults will interact among large software systems, over 100,000 LOC, to see if their interaction types differs from medium or small programs. We also plan to analyze how faults interact inside of the pass/fail independence category at a more fine grained level. We hope to be able to identify features of their interaction which can be isolated thereby providing valuable heuristics to debugging fields. We also plan to further investigate the reason some faults cause obfuscation and others do not.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference, Practice and Research Techniques*, Windsor, UK, September 2007.

[2] H. Agrawal, J. Horgan, S. London, and W. Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of IEEE Software Reliability Engineering*, pages 143–151, 1995.

[3] C. Boogerd and L. Moonen. Assessing the value of coding standards: An empirical study. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 277 –286, 28 2008-oct. 4 2008.

[4] J. S. Collofello and S. N. Woodfield. Evaluating the effectiveness of reliability-assurance techniques. *J. Syst. Softw.*, 9:191–195, March 1989.

[5] V. Debroy and W. E. Wong. Insights on fault interference for programs with multiple bugs. In *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering*, ISSRE '09, pages 165–174, Washington, DC, USA, 2009. IEEE Computer Society.

[6] T. Denmat, M. Ducassé, and O. Ridoux. Data mining and cross-checking of execution traces. A re-interpretation of Jones, Harrold and Stasko test information visualization (Long version). Research Report RR-5661, INRIA, August 2005. Also Publication Interne IRISA PI-1743.

[7] N. DiGiuseppe and J. Jones. On the influence of multiple faults on coverage-based fault localization. In *Proceedings of the 9th ACM/IEEE International Symposium on Software Testing and Analysis*, ISSTA '11, page To Appear, New York, NY, USA, 2011. ACM.

[8] H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 60–70, Washington, DC, USA, 2004. IEEE Computer Society.

[9] H. Do and G. Rothermel. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 141–151, New York, NY, USA, 2006. ACM.

[10] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379 – 1393, 2009. Source Code Analysis and Manipulation, SCAM 2008.

[11] J. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the International Conference on Automated Software Engineering*, pages 273–282, November 2005.

[12] J. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering*, pages 467–477, Orlando, Florida, May 2002.

[13] J. A. Jones, J. F. Bowring, and M. J. Harrold. Debugging in parallel. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 16–26, New York, NY, USA, 2007. ACM.

[14] B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29:155–163, October 1988.

[15] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 2005.

[16] C. Liu and J. Han. Failure proximity: A fault localization-based approach. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 286–295, November 2006.

[17] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *Proceedings of 10th European Software Engineering Conference and 13th Foundations on Software Engineering*, pages 286–295, September 2005.

[18] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 181–190, New York, NY, USA, 2008. ACM.

[19] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5:99–118, April 1996.

[20] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the International Conference on Software Engineering*, pages 465–474, May 2003.

[21] A. Podgurski, W. Masri, Y. McCleese, F. G. Wolff, and C. Yang. Estimation of software reliability by stratified sampling. *ACM Trans. Softw. Eng. Methodol.*, 8:263–283, July 1999.

[22] A. Podgurski and C. Yang. Partition testing, stratified sampling, and cluster analysis. In *Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '93, pages 169–181, New York, NY, USA, 1993. ACM.

[23] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 75–86, New York, NY, USA, 2008. ACM.

[24] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the International Conference on Automated Software Engineering*, pages 30–39, Montreal, Quebec, October 2003.

[25] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *ACM Trans. Softw. Eng. Methodol.*, 13:277–331, July 2004.

[26] G. Rothermel and M. Harrold. Analyzing regression test selection techniques. *Software Engineering, IEEE Transactions on*, 22(8):529 – 551, aug 1996.

[27] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929 –948, oct 2001.

[28] R. Santelices, M. J. Harrold, and A. Orso. Precisely detecting runtime change interactions for evolving software. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:429–438, 2010.

[29] J. Strecker and A. Memon. Relationships between test suites, faults, and fault detection in gui testing. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 12–21, Washington, DC, USA, 2008. IEEE Computer Society.

[30] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[31] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 201–210, New York, NY, USA, 2008. ACM.

[32] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, ICML '06, pages 1105–1112, New York, NY, USA, 2006. ACM.