20th International Symposium on Software Reliability Engineering

Insights on Fault Interference for Programs with Multiple Bugs

Vidroha Debroy and W. Eric Wong
The University of Texas at Dallas
Department of Computer Science
{vxd024000,ewong}@utdallas.edu

Abstract

Multiple faults in a program may interact with each other in a variety of ways. A test case that fails due to a fault may not fail when another fault is added, because the second fault may mask the failure-causing effect of the first fault. Multiple faults may also collectively cause failure on a test case that does not fail due to any single fault alone. Many studies try to perform fault localization on multi-fault programs and several of them seek to match a failed test to its causative fault. It is therefore, important to better understand the interference between faults in a multi-fault program, as an improper assumption about test case failure may lead to an incorrect matching of failed test to fault, which may in turn result in poor fault localization. This paper investigates such interference and examines if one form of interference holds more often than another, and uniformly across all conditions. Empirical studies on the Siemens suite suggest that no one form of interference holds unconditionally and that observation of failure masking is a more frequent event than observation of a new test case failure.

Keywords: multiple faults, fault interference, software testing, fault localization.

1. Introduction

Software fault localization is an actively researched topic in the area of program debugging today. Originally, many fault localization studies demonstrated the effectiveness of their approach, under the assumption that a program would contain only one fault [4,9,14,15,19,22,23]. However, recently the focus has been on the development of fault localization techniques that can be applied to programs with multiple bugs[1] as well [10,16,18,25]. A common way to do this has been to segregate failed test cases into groups or clusters such that the failed test cases in each cluster can be mapped to the same causative fault [6,16,18]. While this may sound deceptively simple, it is in reality, a very difficult thing to do; and one of the reasons is that when two (or more) seemingly independent faults are placed in the same program, they may interact with each other in complex and unpredictable ways.

These interactions may manifest themselves by causing a test case to fail[2] that would normally not have failed due to the presence of any single fault alone. Alternatively, a test case that would normally have failed due to a fault, may no longer fail upon the addition of another fault (to the same program), that *interferes* with the failure-causing effect of the first fault. Multiple faults may of course also exist in the same program, yet not interfere with each other in a manner that is observable by examining test case failure. In such a scenario, the effect of each fault in the program may *seem* to be *independent* of the other.

The objective of this study is to observe such interactions between faults present in the same program, and to empirically deduce if a certain type of interaction holds uniformly or unconditionally. We do this by first asking certain research questions regarding multi-fault interactions, and then trying to address them by means of data collected on multi-fault versions of the seven programs in the Siemens suite [21].

The remainder of this paper is organized as follows: Section 2 details some of the preliminary and background information relevant to understanding this study and Section 3 outlines the experimental design and procedure that is followed. Then, Section 4 presents the empirical study undertaken, while Section 5 contains an analysis and discussion of important issues and considerations regarding the research objectives. Section 6 subsequently overviews research work that is related and relevant to this paper; and finally Section 7 presents the conclusions and future work.

2. Preliminaries

This section first presents a more detailed account of fault interference in multi-fault programs, and provides illustrative examples for each form of interference studied. Additionally, the research questions that are addressed in this paper are described and enumerated to facilitate subsequent analysis.

2.1. Forms of Interference

In wave physics, 'interference' is the phenomenon which occurs when two waves meet while traveling along the same medium. If the amplitudes of the two waves have the same sign, then they add together to form a wave of a larger amplitude. This is termed as 'constructive interference'. On the other hand, if the two amplitudes have opposite signs, then they combine together to form a wave with a lower amplitude and this is termed as 'destructive interference' [7]. We now abstract this terminology and apply it to our research context.

[1] For the purposes this paper, we use fault(s) and bug(s) interchangeably.

[2] Throughout this paper, 'test case failure' refers to the failure of the program to produce the correct output on a test case.

978-0-7695-3878-5/09 $26.00 © 2009 IEEE
DOI 10.1109/ISSRE.2009.14

Given two faults f_1 and f_2, if the presence of both f_1 and f_2 in the same program results in the failure of a test case t that does not fail in the presence of either f_1 or f_2 alone; then this is termed as constructive fault-interference, or simply 'constructive interference'. Correspondingly, given a test case t that fails due to a fault f_1, but no longer fails when another fault f_2 is added to the same program; this is termed as 'destructive interference'. These definitions are however, incomplete without pointing out some important aspects.

Firstly, it is of course possible to observe both phenomena together. The presence of two faults in the same program may cause some test cases to fail - that do not fail due to a single fault alone; yet also cause a test case that fails due to a fault - to now result in a successful execution. Thus, it is possible to observe both constructive and destructive interference simultaneously. Secondly, it is of course also possible to observe neither of the two phenomena i.e. the presence of both the faults may not result in any additional test case failure or the masking of test case failure due to a particular fault. Thirdly, we emphasize that while these definitions are provided with respect to two faults, they are easily extensible to any number of faults present in the same program. Finally, we use the terms 'constructive' and 'destructive' rather loosely in our context. In no way do we imply that more failures are constructive and fewer failures are destructive! These terms are employed here to draw the parallel between fault interference and wave interference and to highlight the concept of faults working together to cause test case failure i.e. working constructively; and faults working against each other to mask test case failure i.e. working destructively.

Let us now go over some simple examples so that we can better understand how and when fault interference (be it constructive or destructive) may occur in a program.

	Correct Program $\mathcal{P}$	$\mathcal{P}$ with fault f_1	$\mathcal{P}$ with fault f_2	$\mathcal{P}$ with fault f_1 and f_2

	read(a,b);
	x=a;
	y=b;
	if((x+y) == 6)
	print(6);
	else
	print(-1);
		read(a,b);
		x=a + 3;
		y=b;
		if((x+y) == 6)
		print(6);
		else
		print(-1);
			read(a,b);
			x=a;
			y=b + 3;
			if((x+y) == 6)
			print(6);
			else
			print(-1);
				read(a,b);
				x=a + 3;
				y=b + 3;
				if((x+y) == 6)
				print(6);
				else
				print(-1);

Test t_1 Input: a = 0 b = 0	output: -1	output: -1	output: -1	output: 6
		Pass	***Pass***	***Fail***

Figure 1 Constructive interference between faults

Figure 1 illustrates constructive interference between faults using a snippet of code. We observe that neither fault f_1, nor fault f_2 alone result in the failure of test case t_1. However, when f_1 and f_2 are placed together in the same body of code, they work together, to cause test case failure in t_1. In contrast, in Figure 2 we observe destructive interference because test case t_1 fails when faults f_1 and f_2 are considered alone; but when these two faults are placed together, the same failing test case is now successful. Note that this is an example of *double destructive interference* because the failure causing effect of either fault has been masked by the presence of both faults. This is not a requirement as by definition, as long as one fault masks the failure causing effect of the other, we have destructive interference; and this need not be a mirror relationship.

	Correct Program $\mathcal{P}$	$\mathcal{P}$ with fault f_1	$\mathcal{P}$ with fault f_2	$\mathcal{P}$ with fault f_1 and f_2

	read(a);
	x=a;
	y=x - 3;
	print(y);
		read(a);
		x=a + 1;
		y=x - 3;
		print(y);
			read(a);
			x=a;
			y=x - 4;
			print(y);
				read(a);
				x=a + 1;
				y=x - 4;
				print(y);

Test t_1 Input: a = 5	output: 2	output: 3	output: 1	output: 2
		Fail	***Fail***	***Pass***

Figure 2 Destructive interference between faults

Finally, Figure 3 constructs a scenario where both constructive and destructive interference can be observed simultaneously. Neither fault f_1 nor fault f_2 can cause failure on test case t_1 alone. However, we have constructive interference when these faults are placed together as now they collectively cause test case t_1 to fail. But on the same set of programs, when we consider test case t_2; the failure causing effect of fault f_1 is masked by fault f_2 when both of them are placed together in the same program. The net effect is for test case t_2 to pass and so we have destructive interference. Thus, on the same set of programs we may observe both constructive and destructive interference. As pointed out before, it may also be possible to neither observe constructive interference nor destructive interference.

	Correct Program $\mathcal{P}$	$\mathcal{P}$ with fault f_1	$\mathcal{P}$ with fault f_2	$\mathcal{P}$ with fault f_1 and f_2

	read(a,b);
	x=a;
	y=b;
	result=0;
	if((x+y) == -7)
	result = 1;
	else
	result = 2;
	print(result);
		read(a,b);
		x=a - 3;
		y=b;
		result=0;
		if((x+y) == -7)
		result = 1;
		else
		result = 2;
		print(result);
			read(a,b);
			x=a;
			y=b - 4;
			result=0;
			if((x+y) == -7)
			result = 1;
			else
			result = 2;
			print(result);
				read(a,b);
				x=a - 3;
				y=b - 4;
				result=0;
				if((x+y) == -7)
				result = 1;
				else
				result = 2;
				print(result);

Test t_1 Input: a = 0 b = 0	output: 2	output: 2	output: 2	output: 1
		Pass	***Pass***	***Fail***
Test t_2 Input: a = -4 b = 0	output: 2	output: 1	output: 2	output: 2
		Fail	***Pass***	***Pass***

Figure 3 Constructive & Destructive interference

While the examples above may be simple, they are used primarily for illustrative purposes to help provide a better picture of fault interference. However, this better understanding prompts us to ask several important questions related to fault interference.

2.2. Research Objectives

In this section we present an informal overview of some important questions that we wish to answer. We then

formalize these questions using set theory and also describe how we address them, in Section 3. Hereafter, we may denote a *research question* simply by 'RQ'.

RQ1: How strong is the independence assumption in a multi-fault program? In order words, is fault interference a rare event?

RQ2: How often can we expect to observe only constructive interference in a multi-fault program?

RQ3: How often can we expect to observe only destructive interference in a multi-fault program?

We may also use combinations of these research questions, some of which are meaningful because they are not addressed by any of the other research questions. We consider RQ2 and RQ3 together to produce RQ4, RQ1 and RQ2 together to produce RQ5, and RQ1 and RQ3 together to produce RQ 6:

RQ4: How often can we expect to observe both constructive and destructive interference in a multi-fault program?

RQ5: How often will the faults in a multi-fault program either be independent or have constructive interference?

RQ6: How often will the faults in a multi-fault program either be independent or have destructive interference?

3. Experimental Design

In this section we present a formal representation of the research questions in the previous section and also present our approach on how to answer them.

Given n faults for a program $\mathcal{P}$, let us denote each fault by f_1, f_2, ..., f_n. Let $\mathcal{P}_i$ denote program $\mathcal{P}$ which now contains fault f_i. Thus, each $\mathcal{P}_i$ ($1 \leq i \leq n$) represents a single fault version of each of the n faults. Also let $\mathcal{P}^M$ denote the program $\mathcal{P}$ with all of the n faults in it together which is the multi-fault version of $\mathcal{P}$. The underlying assumption is that all of these faults can be seeded into the same program without conflict. A similar approach was also used in [11]. Further discussion on how and why the seeding of one fault may conflict with the seeding of another is presented in Section 4.1.

Consider a test set $\mathcal{T}$ for program $\mathcal{P}$. Executing all the test cases in $\mathcal{T}$ on each of the single fault versions of program $\mathcal{P}$, would give us the set of failed test cases for each single fault version - such that each set of failed test cases $\mathcal{S}_i$ corresponds to a faulty version $\mathcal{P}_i$. All of the test cases in $\mathcal{T}$ can also be executed against the multi-fault version $\mathcal{P}^M$ to obtain the set of test cases that fail on the multi-fault version. Let us denote this set by $\mathcal{M}$. Also let $\mathcal{U}$ denote the set of failed test cases obtained by taking the union of each $\mathcal{S}_i$ such that:

$$\mathcal{U} = \mathcal{S}_1 \cup \mathcal{S}_2 \cup ... \mathcal{S}_n$$

The research questions detailed in Section 2 can be answered by observing the relationship between, the set of failed test cases obtained by taking the union of the sets of failed test cases on each single fault version i.e. $\mathcal{U}$, and the set of failed test cases on the multi-fault version $\mathcal{M}$. Each studied relationship between the two sets is listed as follows alongside the research question that it is related to.

(1) $\mathcal{U} = \mathcal{M}$. This means that the two sets are equal which implies that no new test cases failed on the multi-fault version and no test case failures due to any single fault were masked i.e. there was neither constructive, nor destructive interference observed. This observation is equivalent to observing independence. (RQ1)

(2) $\mathcal{U} \subset \mathcal{M}$. This means that each test case that failed due to any single fault still failed on the multi-fault version, but at the same time some new test case failures were observed on the multi-fault version that are not caused by single faults alone. This is equivalent to observing only constructive interference (RQ2).

(3) $\mathcal{M} \subset \mathcal{U}$. This means that no new test case failures were observed on the multi-fault version, but some of the test case failures due to single fault versions were masked. This is equivalent to observing only destructive interference (RQ3).

(4) $(\mathcal{M} \not\subset \mathcal{U}) \wedge (\mathcal{U} \not\subset \mathcal{M}) \wedge (\mathcal{U} \neq \mathcal{M})$. This means that, neither set is a proper subset of the other, yet at the same time they are not equal. This implies that we have some test case failures that have been masked and at the same time some new test case failures that are observed only on the multi-fault version i.e. both constructive and destructive interference (RQ4). Note that it is theoretically possible for $\mathcal{U}$ and $\mathcal{M}$ to not have any intersection at all. This would happen if every test case that fails on the multi-fault version does not fail due to any single fault version; and at the same time any and every test case failure due to a single fault version, is masked in some way in the multi-fault version.

(5) $\mathcal{U} \subseteq \mathcal{M}$. This means that the set $\mathcal{U}$ is either equal to or a subset of $\mathcal{M}$ i.e. either no new test cases fail on the multi-fault version, or some do, but either way there is no masking of failures. This implies we either observe constructive interference or independence, but no destructive interference (RQ5).

(6) $\mathcal{M} \subseteq \mathcal{U}$. This means that the set $\mathcal{M}$ is either equal to or a subset of $\mathcal{U}$ i.e. either no test case failures are masked in the multi-fault version, or some are, but either way there are no new test case failures on the multi-fault version. This implies we either observe destructive interference or independence, but no constructive interference (RQ6).

Having identified these important relationships, we can now observe their frequency of occurrence on a data set, in order to quantify how often each one holds true.

4. Case Study

In order to empirically observe how well each of the relations detailed in Section 3 holds on actual programs, we performed a case study on the seven programs of the Siemens suite [21]. This section describes the subject programs, the manner in which the data was collected and then moves on to discuss the results obtained.

4.1. Subject Programs

The seven programs in the Siemens suite have been well studied and used for several fault localization studies [4,9,10,15,19,22,23,24], and therefore, were an ideal choice for this study as well. The correct and faulty versions of each of the programs as well as the respective test sets were downloaded from [21]. Altogether 132 fault versions were downloaded, but not all of them were useable for this study. Recall in Section 3 that when considering a multi-fault version with *n* faults, we assume that each of the *n* faults can be seeded into the same program without conflict. Unfortunately, this assumption does not hold for several pairs of faults for some programs in the Siemens suite. Some conflicting faults share the same location and thus cannot be seeded simultaneously into the same program. In such a scenario, we randomly discarded one of the faults that conflicted with the position of another. Faults which did not result in any test case failure, such as version 9 of the program schedule2, were also discarded. Table 1 presents a list of the programs, along with the final number of faults and test cases per program used.

Table 1 Summary of the Siemens suite

Program	Number of faults used	Number of test cases
print_tokens	5	4130
print_tokens2	8	4115
schedule	8	2650
schedule2	8	2710
replace	16	5542
tcas	19	1608
tot_info	20	1052

4.2. Data Collection

In order to eliminate any bias, the experiment is best performed using *n*-bug versions across various sizes of *n* (i.e., using multi-bug versions across a broad spectrum of sizes). Thus, for each of the programs we generated multi-fault versions, using different combinations of a varying number of faults. For the programs containing a relatively smaller number of faults, it was possible to exhaustively generate all possible combinations of non-conflicting faults. However, this was not possible for programs with a relatively large number of faults as generating all possible combinations is explosive, and requires massive overhead. To make the experiment more tractable, where the generation of all possible multi-fault versions was not possible (as in the case of programs – replace, tcas and tot_info), we placed a ceiling on the number of multi-fault versions to be used. Then we randomly sampled (without replacement) fault combinations of different sizes until the desired number of multi-fault versions was generated.

Table 2 Multi-fault versions generated per program

Program	Number of multi-fault versions generated
print_tokens	26
print_tokens2	247
schedule	247
schedule2	247
replace	500
tcas	1000
tot_info	1000

Table 2 lists the number of multi-fault versions, i.e. fault combinations, generated for each program. We note that a fewer number of multi-fault versions are generated for replace (500), than for tcas (1000) and tot_info (1000). This is because from Table 1 we observe that the program replace also has a lot more test cases than tcas and tot_info. The time required to perform the experiment is not just determined by the number of multi-fault versions used, but also by the number of test cases that need to be executed on the programs. The experiment requires that for each of the multi-fault, and each of the single fault versions of each program, the entire test set be executed against the versions to obtain the set of failed test cases. Thus, since the test set for the program replace consists of 5542 test cases; to reduce the execution overhead involved, we generated 500 multi-fault versions instead of 1000. Adding up the entries in the second column of Table 2, we find that a total of 3267 multi-fault versions (with a varying number of faults per version) were used in the experiment.

In order to deduce that a test case had failed on a faulty version, we executed it against the faulty version and compared the output to the output obtained by running the same test case against the corresponding correct version of the program. If the outputs differed, then the execution of the test case on the faulty version was said to have resulted in 'failure'; and if the outputs were the same, then the test case execution was deemed to have been 'successful'. This is consistent with the taxonomy in [1] where a failure is defined as an event that occurs when a delivered service deviates from correct service. All program executions were on a PC with a 2.13GHz Intel Core 2 Duo CPU and 8GB physical memory. The operating system was SunOS 5.10 (Solaris 10) and the compiler version used was GCC 3.4.3.

4.3. Results

In this section we present the results of the case study that was conducted on the Siemens suite. Table 3 presents a summary of the frequencies of occurrence of each of the relations that were proposed in Section 3. Please note that the data presented in the table corresponds to the data collected across all of the multi-fault versions generated i.e. all 3267 data points. The data collected is not strongly indicative that any of the studied relationships holds uniformly throughout. About a third of the time (33.12%) we observe that the independence assumption holds because the multi-fault

version does not mask failures, nor does it cause previously successful test cases to fail. This means that about 66.88% of the time we expect to see interference of some kind. However, this percentage is too low to deem that fault interference will almost always occur, even though it does indicate that fault interference is a fairly frequent event.

The highest frequency of occurrence for any of the studied relationships is on the relation $\mathcal{M} \subseteq \mathcal{U}$ (47.72%) which corresponds to RQ5. This means that almost half of the time, either the two sets are equal i.e. there is neither destructive nor constructive interference; or the interference is only destructive i.e. we have masking. In contrast, the lowest frequency of occurrence is for the relation $\mathcal{U} \subset \mathcal{M}$ (5.05%) which corresponds to RQ2. This means that only a very small percent of the time can we expect to observe only constructive interference, and this seems to be a much rarer event than some of the other relationships. However, this only provides us with a macroscopic idea of what is really going on because we consider all multi-fault combinations (even though they may contain a different number of faults) collectively in the same pool. Perhaps there is a hidden link between the frequency of each relationship and the number of faults that may be seeded into a multi-fault version. With this in mind, we decided to extend our analysis such that we observe the occurrence of each relationship across different multi-fault versions with varying numbers of faults in them.

Table 4 presents the data for each of the relations based on the number of faults present in each of the multi-fault versions. Once again we do not observe the occurrence of any of the relations uniformly; but do however, observe some interesting trends.

Firstly, we observe, as we did with the collective analysis (Table 3), that the frequency of $\mathcal{U} \subset \mathcal{M}$ is relatively lower than that of some of the other relations. In fact, the occurrence of $\mathcal{U} \subset \mathcal{M}$ is always less than or equal to that of $\mathcal{M} \subset \mathcal{U}$ and this seems to be independent of the number of faults in the multi-fault version (RQ2 and RQ3).

Table 3 Frequency of occurrence of each relation across the entire data set

Number of Multi-fault versions	$\mathcal{U} = \mathcal{M}$ (RQ1)	$\mathcal{U} \subset \mathcal{M}$ (RQ2)	$\mathcal{M} \subset \mathcal{U}$ (RQ3)	$\mathcal{U} \subseteq \mathcal{M}$ (RQ5)	$\mathcal{M} \subseteq \mathcal{U}$ (RQ6)	$(\mathcal{M} \not\subset \mathcal{U}) \wedge (\mathcal{U} \not\subset \mathcal{M}) \wedge (\mathcal{U} \neq \mathcal{M})$ (RQ4)
3267	1082 (33.12%)	165 (5.05%)	477 (14.6%)	1247 (38.17%)	1559 (47.72%)	1543 (47.23%)

Table 4 Frequency of occurrence of each relation based on the number of faults in each multi-fault version

Num. Faults in a multi-fault version	Num. versions generated	$\mathcal{U} = \mathcal{M}$ (RQ1)	$\mathcal{U} \subset \mathcal{M}$ (RQ2)	$\mathcal{M} \subset \mathcal{U}$ (RQ3)	$\mathcal{U} \subseteq \mathcal{M}$ (RQ5)	$\mathcal{M} \subseteq \mathcal{U}$ (RQ6)	$(\mathcal{M} \not\subset \mathcal{U}) \wedge (\mathcal{U} \not\subset \mathcal{M}) \wedge (\mathcal{U} \neq \mathcal{M})$ (RQ4)
2	96	78.13%	7.29%	12.50%	85.42%	90.63%	2.08%
3	184	52.72%	16.30%	23.37%	69.02%	76.09%	7.61%
4	253	36.36%	18.97%	24.11%	55.34%	60.47%	20.55%
5	247	28.34%	14.17%	18.22%	42.51%	46.56%	39.27%
6	231	25.97%	9.96%	10.39%	35.93%	36.36%	53.68%
7	269	26.77%	3.72%	10.41%	30.48%	37.17%	59.11%
8	353	29.46%	1.98%	8.50%	31.44%	37.96%	60.06%
9	431	30.39%	0.93%	7.66%	31.32%	38.05%	61.02%
10	397	27.20%	0.00%	10.58%	27.20%	37.78%	62.22%
11	370	33.78%	0.00%	14.86%	33.78%	48.65%	51.35%
12	241	34.02%	0.00%	22.41%	34.02%	56.43%	43.57%
13	119	35.29%	0.84%	21.01%	36.13%	56.30%	42.86%
14	47	38.30%	0.00%	25.53%	38.30%	63.83%	36.17%
15	25	20.00%	0.00%	44.00%	20.00%	64.00%	36.00%
16	2	50.00%	0.00%	0.00%	50.00%	50.00%	50.00%
17	2	0.00%	0.00%	100.00%	0.00%	100.00%	0.00%

Secondly, we find that the relation $\mathcal{U} = \mathcal{M}$ seems to decrease at first when the number of faults in a faulty version is increased, but then this decrease tapers gradually and actually switches to a small increase subsequently. The rapid initial decrease is at least suggestive that while the independence assumption may hold for a multi-fault version with a relatively smaller number of faults; it does not hold as strongly once the number of faults is increased (RQ1).

The trends between $\mathcal{U} \subset \mathcal{M}$ and $\mathcal{M} \subset \mathcal{U}$ are easier observed visually and therefore, we present these two relations in a graph in Figure 4. The trend for $\mathcal{U} = \mathcal{M}$ is also

presented. For clarity, the other relationships are not represented in this graph.

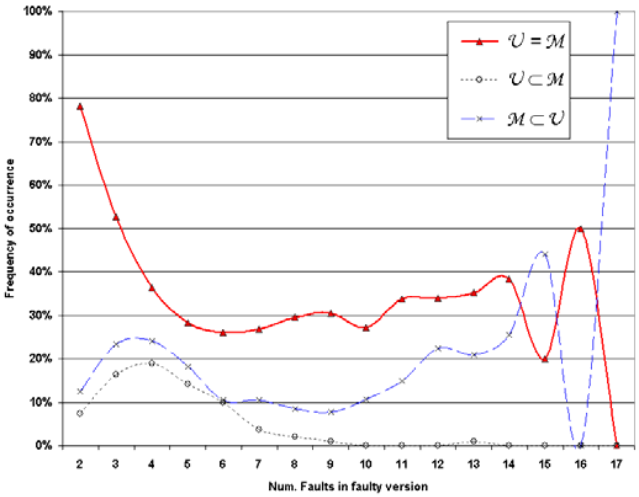

Figure 4 Frequency of relations with varying number of faults

Based on the graph, once again we observe that the frequency of occurrence of the relation $\mathcal{U} \subset \mathcal{M}$ is always less than that of $\mathcal{M} \subset \mathcal{U}$ and this does not change even as we increase the number of faults. Additionally, while the two curves seem to follow the same pattern initially (n=6), they subsequently move on to follow two completely independent paths.

From Table 4 we can also observe that the frequency at which both constructive and destructive interference occurs increases as the number of faults in a multi-fault program increases; peaks at 62.22%, and then starts to decrease even though the number of faults is still increased. Based on Table 3 however, we note that almost half of the time (47.23%) we can expect to see both constructive and destructive interference. This means that observing both types of interference together, is definitely not a rare event (RQ4).

We point out that while data has also been provided for multi-fault versions that can contain as many as up to 17 faults; the number of samples for the versions with 16 and 17 faults, are only 2 each. Therefore, the data for multi-fault versions of these sizes may not be representative and derived results maybe subject to bias. Further discussion on this, and an explanation regarding the small sample-sizes, is provided in Section 5.

5. Discussion

In this section, we present a discussion on some other pertinent issues that are relevant to this study and also discuss the threats to validity.

5.1. Additional Factors to Consider

In addition to the number of faults that are present in a multi-fault program, there may be other impacting factors that affect the occurrence of the studied relationships.

Given two programs of different sizes (a different number of executable statements), but with the same number of faults in each program, the larger sized program shall consequently have a lower fault density (with respect to all of the executable statements in the program). A lower fault density would in turn imply that there is a higher probability of faults acting independent of one another than in the case of a program with a higher density. This is because in the case of more executable statements, the faults themselves have a higher chance of being more spread out and perhaps in completely different blocks[3]. Thus, the size of a multi-fault program may affect the interactions between the faults and consequently may affect the frequency of the relations.

Each branch present in a program represents an alternate flow of control in that program. Therefore, the more branching that is present in a multi-fault program, the higher the likelihood that the faults might be in separate branches, and consequently the higher the probability of the faults acting independent of one another. With this in mind we decided to try and observe if the relative sizes of each of the programs under study; or the number of branches in each of the programs, had any specific effect on the relations under study. Table 5 presents the size of each of the programs in the Siemens suite in terms of the number of executable statements in the program; as well as the number of branches in each program.

Table 5 Size and number of branches for studied programs

Program	Size of program (executable statements)	Number of branches
print_tokens	175	109
print_tokens2	178	162
schedule	121	66
schedule2	112	88
replace	216	176
tcas	55	50
tot_info	113	68

Size of the program: As described above, intuition suggests that the higher the number of executable statements in a program, the higher the likelihood of observing independence between faults. Thus, we expect to see an increase in the frequency of occurrence of the relation $\mathcal{U} = \mathcal{M}$ with an increase in the size of a program. Figure 5 graphically shows the observed trends on all of the relations except $\mathcal{U} \subseteq \mathcal{M}$ and $\mathcal{M} \subseteq \mathcal{U}$. These relations have been left out to maximize clarity and because they can be derived easily from the relations $\mathcal{U} = \mathcal{M}$, $\mathcal{U} \subset \mathcal{M}$ and $\mathcal{M} \subset \mathcal{U}$.

But contrary to our intuition, based on Figure 5, we find that there is no evidence to support the claim that the independence assumption is especially stronger in the case of programs with a relatively smaller size (in terms of the number of executable statements). The curve is somewhat

[3] Such an assumption is strictly probabilistic. In the context of software, faults may not be distributed uniformly but rather may cluster together in certain parts of the code.

haphazard and does not follow a set pattern. Additionally, the same observation can be made of the rest of the studied relations.

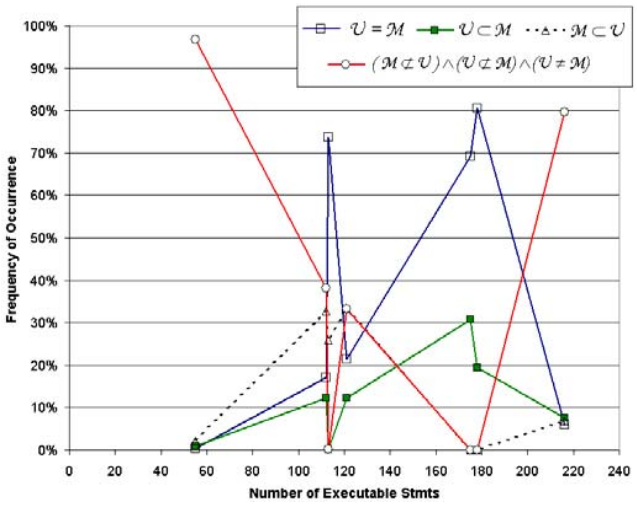

Figure 5 Frequency of relations with respect to number of executable statements

Number of branches in the program: As with program size, our intuition suggests that with an increase in the number of branches in the program, we should expect to see and increase in the frequency of occurrence of the independence assumption. Figure 6 plots the curves of the relations against the number of branches in the programs in a manner similar to Figure 5. Also, for the same reasons as Figure 5, the curves for the relations $\mathcal{U} \subseteq \mathcal{M}$ and $\mathcal{M} \subseteq \mathcal{U}$ have not been plotted.

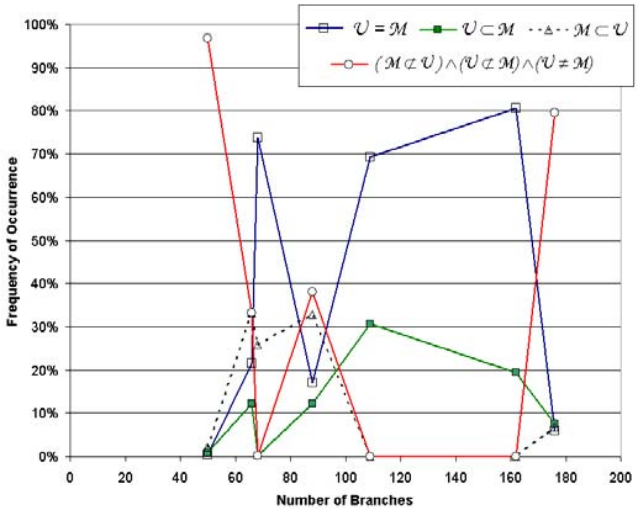

Figure 6 Frequency of relations with respect to number of branches

Once again, as with Figure 5, we observe that there does not seem to be a set pattern to how the occurrence of the relations changes with the number of branches in the program. Moreover, we are unable to draw the conclusion that the strength of the independence assumption increases as the number of branches in a program increases. The graphs themselves can be a little difficult to make out and indeed, visualization alone does not seem to be enough here. Therefore, in Table 6 we also represent the data in terms of the correlation coefficients with respect to the number of branches and statements in the programs, for each of the studied relations.

Table 6 Correlation between relations and number of branches and executable statements

Relation	Pearson Correlation Coefficient (r)	
	# Branches	# Ex. Stmts
$\mathcal{U} = \mathcal{M}$	0.20	0.30
$\mathcal{U} \subset \mathcal{M}$	0.39	0.53
$\mathcal{M} \subset \mathcal{U}$	-0.48	-0.35
$(\mathcal{M} \not\subset \mathcal{U}) \wedge (\mathcal{U} \not\subset \mathcal{M}) \wedge (\mathcal{U} \neq \mathcal{M})$	-0.09	-0.27

The first column in the table represents each of the relations depicted in Figures 5 and 6, and the second and third columns provide the correlation coefficients between the frequency of the relations and the number of branches and executable statements, respectively. None of the relations exhibits a strong positive correlation either with respect to the number of branches, or with respect to the number of executable statements. In fact, negative correlations are observed for $\mathcal{M} \subset \mathcal{U}$ and $(\mathcal{M} \not\subset \mathcal{U}) \wedge (\mathcal{U} \not\subset \mathcal{M}) \wedge (\mathcal{U} \neq \mathcal{M})$. The relation $\mathcal{U} \subset \mathcal{M}$ does show some positive correlation with respect to the number of executable statements (r = 0.53). However, it is not nearly strong enough to be considered significant. A lack of strong correlation (whether positive or negative) implies that it is impossible to conjecture a possible relationship between the variables under analysis.

5.2. Implications of the Results

While the results obtained from the experiment may not positively conclude that any one of the studied relations is always observable across the various programs, it does lead to some important implications:

(1) The fact that the relation $\mathcal{U} = \mathcal{M}$ does not hold uniformly implies that fault localization studies should not make the independence assumption when faults are placed together in the same multi-fault program. Not just does the independence assumption not always hold for multi-fault programs with a small number of programs, but rather it suffers from certain degradation as the number of faults is increased.

(2) If the independence assumption clearly does not hold, then it means interference of some kind is occurring. Furthermore, while interference is not always observed, in our studies it is observed approximately two-thirds of the time which implies that it is a fairly frequent event. Fault localizers must be prepared to specifically handle this

interference without suffering from a significant loss in effectiveness.

(3) Based on our results we observe that the relation $\mathcal{M} \subset \mathcal{U}$ is far more frequent than $\mathcal{U} \subset \mathcal{M}$ and that this trend between the two relations is almost constant irrespective of the number of faults that may be present in the multi-fault program. This means that we observe constructive interference alone less than we observe destructive interference alone, and this in turn means that the number of test cases that fail on a multi-fault version may not be many. Fault localizers that work with dynamic information, need to be able to maintain effectiveness even with less available information and need to be able to assess the importance of each test case when localizing faults such as in [22].

(4) Even though constructive interference alone occurs less than destructive interference alone, there are still several instances where the presence of multiple faults in the same program have indeed led to the failures of test cases that would not normally have failed due to any one fault alone. From Table 4 we observe that constructive interference alone does not seem to occur much in programs with a relatively larger number of faults, but rather occurs in programs with a smaller number of faults. However, from Table 3 we observe that it occurs alone at least about 5% of the time. This means that fault localizers often need to be able to distinguish test cases that fail due to several faults from test cases that fail due to just one fault; and consequently, need to link the test case failure to all of its causative faults, as opposed to just linking the failure to one of them. As pointed out by the authors of [25], when performing fault localization, a given test case run/execution can exhibit more than one bug.

(5) A very important observation is that the relationship $(\mathcal{M} \not\subset \mathcal{U}) \wedge (\mathcal{U} \not\subset \mathcal{M}) \wedge (\mathcal{U} \neq \mathcal{M})$ seems to hold about half of the time (47.23%); which means that constructive and destructive interference are both observed on the same multi-fault program. This implies that while fault localizers need to be able to handle constructive and destructive interference; they cannot just focus on only one kind of interference at a time. The fault localizer needs to be designed such that its effectiveness is not adversely affected when both forms of interference are observed together.

5.3. Threats to Validity

In this section we discuss some of the potential threats to validity of our approach, and therefore, threats to the subsequent results and implications that have been derived.

Since this is an initial study, only an analysis on the seven programs of the Siemens suite has been performed. These programs are of a relatively smaller size and may not be representative of much larger programs. Consequently, the degrees to which the various studied relationships hold may vary when the analysis is extended to other programs; and we may not be able to generalize our results across all of them. However, we did not choose the programs of the Siemens suite because they are small in size. Rather, our choice is validated by the fact that these programs have been used extensively in many different fault localization studies [4,9,10,15,19,23,24]. Since our study has direct implications for such fault localization research; we felt it best to analyze and derive results on the same sets of programs that had been used in those studies. It is also important to note that test cases and the nature of the faults used in our study have an impact on the results and conclusions.

Some of the sample sizes that are used to derive results in this study may not be sufficiently large. We would have liked to have had more multi-fault versions that contained a relatively high number of faults. For example, we only had 2 multi-fault versions each that contained 16 or 17 faults. However, such considerations are more governed by the availability of faults, and less by our own choices. Based on Table 1 we observe that 4 out of the 7 programs studied, did not have more than 8 simultaneously seed-able faults to begin with; and therefore, we could not generate multi-fault versions of these programs that contained a high number of faults in them. Thus, as mentioned before in Section 4, we recognize the fact that some of the results obtained may be subject to certain bias due to small sized samples.

It is also important to recognize the fact that test case failure is dependent on, and maybe linked to, the environment in which the execution takes place. It is possible for the same test set to be executed, against the same program but in different environments, and result in two different sets of test case failures. Such phenomenon has been reported in fault localization studies where one study may make use of a fault that has not been used by another study. This is sometimes because no test case failure was observed on that particular fault by one study (rendering it unsuitable for dynamic based fault localization research), but test case failure was observed on that same fault by the other study, which allowed its use. For example, one of the faults of the program *replace* in the Siemens suite is left out in [15], yet the same fault is useable in studies such as [23,24]. However, the experiments performed in this study have all been performed under the same environment and therefore, the results are consistent with respect to each other.

6. Related Work

In this section we provide an overview of work that is related and relevant to the research presented in this paper.

Several fault localization studies have discussed programs with multiple bugs as well, and they have usually attempted to group failed executions such that the failed test cases in each group correspond to the same causative fault [6,16,18]. However, as recognized by the authors of [16], the 'due-to' relationship between failed cases and underlying faults is unknown without manual investigation and an ideal partitioning is generally unachievable. This paper does not present any new fault localization technique, but rather investigates this 'due-to' relationship and researches how often a failed test cannot just be traced back to its causative fault, but instead must be traced back to its causative *faults*.

But fault localization is not the only research area where one might consider and explore the assumption of independence between faults. In [5] the authors explore specifically the effect of assuming a non-uniform probability distribution of faults in chips; but also discuss independence of faults. They observe that the assumption of independence of faults is unrealistic in practical cases. The fault-independence assumption is also made in [8] and [12], and in [12] the assumption is made of component failures, in the context of model-based hardware diagnosis.

Independence of failures, or simply failure-independence, is also a heavily researched topic in the area of software reliability. Several Non-Homogenous Poisson Process Software Reliability Growth Models (NHPP SRGMs) assume software failures occur at random, and are independent of each other [2]. Software reliability often makes use of Markov models and one of the importance features of a Markov model is that the transition from state *i* to another state depends only on the current state [17]. Thus, in [20] the authors assume that when a service is called, its reliability does not depend on methods previously called, and so failures of services are independent. Multi-version of N-version programming too assumes that programs that have been developed independently, shall fail independently [3]. However, experiments by [13] suggest that such an assumption cannot always be made and that N-version programming must be used with care and that the analysis of its reliability must include the effect of dependent errors.

In the context of regression testing, the authors of [11] investigate the costs and benefits of several regression test selection (RTS) strategies when the number of changes between the base and subsequent versions of a program increases. They model varying amounts of modifications by seeding multiple mutually independent faults into the same base program. The authors hypothesize that a test suite that reveals a fault in a program (when it is the only fault in the program) might no longer reveal the same fault when it is mixed with other faults in the same program. The authors report that their data is consistent with the hypothesis that fault detection effectiveness decreases as the number of faults increases.

Thus, the independence assumption is certainly not a new concept and has been employed several times in several different contexts. Our results on the Siemens suite are suggestive that such an assumption is not a sound one to make, and that approximately two-thirds of the time, such an assumption would be incorrect. To the best of our knowledge, we are not aware of any other research work that shares an identical objective and performs the experiment on the Siemens suite in a manner identical to the one in this study.

7. Conclusions and Future Work

This paper investigates the interactions that might take place between multiple faults, present in the same program, and how these interactions may manifest themselves to cause, or mask, test case failure.

We do so, by first asking questions about the relationships between the set of test cases that fail on a multi-fault version; and the set of failed test cases obtained by taking the union of the sets of failed test cases from each of the corresponding single-fault versions of the multi-fault program in question. Then we observe how strongly these relations hold (how frequently they occur) by sampling different fault combinations from the programs of the Siemens suite, and then observing and analyzing test case failures.

Results are suggestive of the fact that no one relationship seems to hold unconditionally across all of the studied programs; and that the fault-independence assumption seems to be fairly weak in that it holds only about a third of the time. Furthermore, destructive interference seems to be more common than constructive interference because test case failure seems to be masked more often than it is caused, by the seeding of different faults together in the same program. In addition, it is quite common for these forms of interference to occur simultaneously and this was observed in almost half of the cases.

Future work includes, but is not limited to, extending our analysis to analyze fault interference on different programs of varying sizes. Also, the analysis performed in this study observes when the presence of multiple faults in a program results in test case failure or masking, different from what would have occurred had we only considered the corresponding single-fault versions. We also wish to observe when the addition of an n^{th} fault causes different behavior from an *(n-1)* fault program. For example, the addition of a 3^{rd} fault into a program may cause considerable difference from each of the three corresponding single-fault versions; but also from the different combinations of 2-bug versions. Additionally, we have analyzed the various trends between fault interactions in terms of simple factors such as the number of faults in multi-fault version; the size of the programs and the number of branches in the programs – and have concluded that they have little impact on the various studied interference relations. Further insights may be revealed by taking into account more complex factors, or perhaps combinations of these simple factors.

8. Acknowledgment

The authors wish to thank Hyung Jae Chang of the Software Technology Advanced Research (STAR) Lab at the University of Texas at Dallas for his help in preparing this paper.

References

1 A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, 1(1):11-33, January 2004

2 W. R. Blischke and D. N. P. Murthy (*Editors*), "*Case studies in reliability and maintenance*", John Wiley & Sons, 2003

3 L. Chen and A. Avizienis, "N-version Programming: A fault-tolerance approach to reliability of software operation," Digest of Papers FTCS-8: *The 8th Annual International Conference on Fault Tolerant Computing*, Toulouse, France, pp. 3-9, June 1978

4 H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th Internationa*l *Conference on Software Engineering*, pp. 342-351, St. Louis, Missouri, May 2005

5 F. Corsi, C. Marzocca and S. Martino, "Assessing the Quality Level of Digital CMOS IC's under the Hypothesis of Non-Uniform Distribution of Fault Probabilities," in *Proceedings of the European Design and Test Conference*, pp. 72, Paris, France, March 1996

6 W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles" in *Proceedings of the 23rd International Conference on Software Engineering*, pp. 339–348, Canada, May 2001

7 D. Halliday, R. Resnick and J. Walker, "*Fundamentals of Physics Extended*," 8th Edition", John Wiley & Sons, 2008

8 V. S. Iyengar and D. T. Tang, "On simulating faults in parallel", in *Proceedings of the 18th International Symposium on Fault Tolerant Computing*, pp. 110-115, Tokyo, Japan, June 1988

9 J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 273-283, California, November 2005

10 J. A. Jones, J. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 16-26, London, UK, July 2007

11 J. M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test application frequency," in *Proceedings of the 22nd International Conference on Software Engineering*, pp. 126-135, Limerick, Ireland, June 2000

12 J. Kleer and B. C. Williams, "Diagnosing Multiple Faults", *Artificial Intelligence*, 32(1):97-130, April 1987

13 J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multi-version programming", *IEEE Transactions on Software Engineering*, 12(1):96-109, January 1986

14 B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 15-26, Chicago, June 2005

15 C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: a hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, 32(10):831-848, October 2006

16 C. Liu and J. Han, "Failure proximity: a fault localization-based approach," in *Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 286-295, Oregon, November 2006

17 M. R. Lyu (*Editor*), "*Handbook of Software Reliability Engineering*", IEEE Computer Society Press, 1996

18 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 25th International Conference on Software Engineering*, pp. 465-475, Oregon, May 2003

19 M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pp. 30-39, Montreal, Canada, October 2003

20 R. H. Reussner, H. W. Scmidt and I. H. Poernomo, "Reliability prediction for component-based software architectures" *Journal of Systems and Software* 66(3): 241-252, June 2003

21 Siemens Suite, http://www-static.cc.gatech.edu/aristotle/Tools/subjects, January 2007

22 W. E. Wong, Y. Qi, L. Zhao and K. Y. Cai, "Effective fault localization using code coverage", in *Proceedings of the 31st Annual International Computer Software and Applications Conference*, pp. 449-456, Beijing, China, February 2007

23 W. E. Wong and Y. Qi, "BP neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, 19(4), June 2009. An earlier version appeared in *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering*, pp. 374-379, Boston, Massachusetts, USA, July 2007

24 W. E. Wong, T. Wei, Y. Qi and L. Zhao, "A Crosstab-based statistical method for effective fault localization, in *Proceedings of the 1st International Conference on Software Testing, Verification and Validation*, pp. 42-51, Lillehammer, Norway, April 2008

25 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*, pp. 1105-1112, Pittsburgh, Pennsylvania, June 2006