The Failure Behaviors of Multi-faults Programs: An Empirical Study

Yan Xiaobo, Liu Bin* , Li Jianxing
Science & Technology on Reliability & Environment Engineering Laboratory,
Beihang University, Beijing, 100191, China
(yxbbuaa, liubin, ljx)@buaa.edu.cn

Abstract—Multi-faults contained in a program can manifest themselves as unexpected failure behaviors. Understanding the failure behaviors of multiple faults is essential for enhancing strategies of program debugging, software fault detection and software maintenance. This paper presents an empirical study of the failure behaviors of multi-faults for four industrial software systems. These four software systems contain a total of 128 software faults, which are identified by 3111 test cases. Our observations show that: 1) independent assumption holds true in most cases; 2) less than 3 faults in multi-faults programs are concurrently triggered by one test case; 3) the failure behaviors of multi-faults can mainly be accounted by dominant faults; 4) the interactions of the dominant faults and recessive faults can cause unexpected behaviors of the system.

Keywords—software multi-faults; failure behavior; fault location.

I. INTRODUCTION

Understanding the nature of software faults is fundamental in software engineering[1] for various purposes, such as defect prevention[2, 3], defect prediction[4], fault localization[5], so as to comprehensively improve software reliability[1], safety[6] and dependability[7]. As one aspect of software fault nature, how multiple faults interact to manifest themselves as new failure behaviors is significant for designing program debugging strategies. There are many fault localization methods, such as slice-based[8], spectrum-based techniques[9]. Most fault localization methods assume that each program has a single bug[5]. However, the empirical studies in[10, 11] imply that individual failures are often triggered by multiple bugs spread throughout the system. Digiuseppe and Jones[12] find that multiple bugs have a negligible effect on the effectiveness of the fault localization.

There are two common ways to localize multiple faults. One way assumes that different faults in the same program fail independently. A representative theory under this assumption is Bayesian reasoning framework for software fault location[13-15]. The other way is to use clustering algorithm to group failed test cases into different fault-focusing clusters, and the failed test cases in the same cluster are related to the same bug[16]. These methods assume that program components fail independently, which may not hold in practice.

Researchers have also been studying the nature of multi-faults and their interactions[12, 17, 18]. The general paradigm of these studies is first to choose “perfect” program versions without any faults; then, inject controllable faults into theses ‘perfect’ versions to get single-fault versions and multi-faults versions; finally compare the differences of execution results between the single-fault versions and multi-faults versions. Through fault injections, researchers find that the interferences between multi-faults is complicated[17, 18]. No one form of interference holds unconditionally, and the phenomenon of failure masking has been frequently observed[17].

Fault injection based methods have the advantage of studying multi-faults in a controllable way. However, the injected faults may hard to enumerate all the possible faults and their combinations. Furthermore, faults occurred in real industrial systems may difficult to expect. It is interesting to extract the failure behaviors of multi-faults programs used in industry if historical data are available.

This paper aims to explore the failure behaviors of multi-faults programs through empirical investigation on four programs that have been running in the Chinese Railway System. The exploratory results can be further used as a guidance for modeling or assessing failures for multi-fault programs.

The remainder of this paper is organized as follows: Section II reviews relevant works. Section III proposes the research questions that guide the empirical study. Section IV presents the method. Section V presents the results, including an analysis and implication of important issues corresponding to the result. Section VI discusses the findings with current researches and Section VII makes conclusion.

II. RELATED WORK AND BACKGROUND

In study[17], the authors find that fault interferences tend to either trigger or mask some execution failures. Software faults can then be divided into constructive interference (when execution fails in the presence of two bugs in the same program) but does not is the presence of either bug alone) and destructive interference (when execution fails due to a bug but no longer fails when another bug is added to the same program). Furthermore, Digiuseppe et al.[18] investigated four significant types of interactions between multi-faults inspired by Debroy and Wang’s idea[17], and concluded that the interaction type “faults obscuring the effects of other faults” is the most prevalent type.

In both study[17] and study[18], the research paradigm is comparing the test cases between single-fault versions and multi-faults versions. For example, considering a test set T for program P. Execute all the test cases in T on each of the single-fault versions of program P, get the set of failed test cases $S_i$
corresponding to a faulty version $p_i$. For multi-faults versions, run the test cases in $T$ and obtain the failed test cases. Then, collect test cases data, compare the union of each $S_j$ and the failed test cases in multi-faults versions, so as to get the results about the interactions between single faults and multi-faults.

From the perspective of test case execution, the above classification can be effectively used in practice. However, some faults can be covered up in this testing phase. There could be some defects that will not or seldom lead to failures with given inputs. These faults are all ‘silent’ faults, which means we cannot observe them from the perspective of the execution of the test case.

On the other hand, we have to deal with the problem of failure masking. Supposing there is a data processing error in a program, meanwhile, a memory overflow error is also triggered by the same test case. Under such circumstances, we can only observe the failure of memory leak. The fault of data processing is masked by the fault of memory leak. Thus, the result of test case will cover some faults in these occasions. Especially in real testing environments, testing results are often highly integrated with multiple functions or modules’ outputs.

To further investigate the interferences of multi-faults from the perspectives of failure mode and failure manifestation, the following definitions are given:

Given $n$ faults in a program $P$, let us denote each fault by $f_1$, $f_2$, ..., $f_n$. Let $p_i$ denote program $P$ which contains fault $f_i$ only and $P^m$ denote the program $P$ with $m$ faults. For every single-fault version $p_i$ ($1 \leq i \leq n$), let $F_i$ represent the failure manifestation of fault $f_i$. For multi-faults version $P^m$, $F(P^m)$ represents the failure manifestation of $P^m$. If we denote set $\gamma$ as the union of each failure manifestation $F_i$ of each single fault $f_i$ in the program $P^m$:

$$\gamma = F_1 \cup F_2 \cup \cdots \cup F_m$$

Similar to study [11], there will be three interferences:

- $F(P^m) = \gamma$. This means the failure of multi-faults is the result of independent interactions of $F_i$. We can call this interaction as independent-failure.
- $F(P^m) \subseteq \gamma$. This means the failure of multi-faults represents a ‘destructive’ interference among multi-faults. In other words, failure-masking is activated.
- $\exists f \in F(P^m), f \notin \gamma$. This means the failure of multi-faults shows new failure mode or failure manifestation compared to $F_i$. In this paper, we call it failure-emerging.

For ‘silent’ fault, in testing phase, this kind of faults cannot be manifested in normal conditions, such as document faults and redundant code. Thus, in previous research, we take few focus on this kind of faults. However, in multi-faults program, will it still be ‘quiet’? In this paper, we want to study the interactions of this type of defects from the real data of industry software.

In the field of genetic engineering, there are two kinds of genes, dominant gene and recessive gene. Dominant gene will manifest its genetic information in organism. Recessive gene will manifest its genetic information only when there is no dominant gene. We now abstract this terminology and apply it to our research context.

Learning from gene theory, the faults can be ‘dominant’ or ‘recessive’.

Given fault $f_i$ in module $i$. Fault $f_i$ is recessive if there is no test case that can lead to failure. Correspondingly, fault $f_i$ is dominant if there is at least one test case that can lead to failure. In formal analysis, consume $T = \{t_1, t_2, ..., t_n\}$ is the test set for module $i$, let $f(t_i)$ indicate execution of the test case $t_i$, then we can conclude that:

- $f_i$ is dominant if:
  $$\exists t_0, t_1 \in T, f(t_1) = \text{pass}$$
- $f_i$ is recessive if:
  $$\forall t_0, t_1 \in T, f(t_1) = \text{fail}$$

What should be noted is that in this paper, we only discuss the functional testing of the system. Which means the definitions above is focused on the functional test cases for software systems.

III. RESEARCH QUESTIONS

This study aims to gain empirical evidence for the following questions:

- **RQ1**: How often does the independence assumption in a multi-faults program hold? In other words, what is the proportion of independent-failure ($F(P^m) = \gamma$) in failed test cases?

- **RQ2**: Do interactions between multi-faults cause new failure behaviors? If so, what is the proportion of failure-emerging ($\exists f \in F(P^m), f \notin \gamma$) in failed test cases?

- **RQ3**: Do the interactions between multi-faults mask or cover some failures’ manifestation? If so, what is the proportion of failure-masking ($F(P^m) \subseteq \gamma$) in failed test cases?

- **RQ4**: What is the distribution characteristics of the dominant and recessive faults?

- **RQ5**: What is the effect of dominant and recessive faults on failure behaviors?

IV. METHOD

In order to identify the differences of failure behaviors between single faults and multi-faults, we investigated the test data from four representative industrial software systems. In this section, we provide the details of the empirical data and analysis process.

**A. Data Collection**

The defect data in our study comes from the Computer Software Reliability Assessment and Testing Centre (CATC), the same database used in [19]. The CATC is a third-party software testing institution, which has been conducting acceptance tests for Aviation Industry Corporation of China...
(AVIC) for a long time. The CATC has a software defect database including a large set of detailed defect reports for avionics software systems. All the defects included in the database had been reviewed and confirmed jointly by test engineers, developers, experts and customers. Moreover, CATC has a rigorous quality assurance process to ensure the accuracy and completeness of defect reports. Therefore, the data validity can be guaranteed in our experiments.

In order to analyze the precise failure manifestations of multi-defects, we reviewed requirements, design documents and source codes together. This empirical analysis was performed on 4 complex systems, which contains a total of 128 defects. Table 1 shows the basic information for these four software systems.

<table>
<thead>
<tr>
<th>Name</th>
<th>Lines of code</th>
<th>Programming language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver</td>
<td>35658</td>
<td>C</td>
</tr>
<tr>
<td>Tools</td>
<td>52672</td>
<td>C++</td>
</tr>
<tr>
<td>Specificlib</td>
<td>49912</td>
<td>C</td>
</tr>
<tr>
<td>Workspace</td>
<td>19775</td>
<td>C</td>
</tr>
</tbody>
</table>

The four programs are core components of a general software development platform which is designed for a railway system. Sub-system driver is the platform driver whose function includes diagnostic storage driver, nonvolatile storage driver, hardware IO control driver and direct IO control driver. Program Tools is responsible for plug-in tool management. Specificlib is the basic function module library, which consists of algorithm function library, control function library and diagnostic function library. The last program Workspace provides underlying service for the whole platform, which includes start control services, interactive response services and task scheduling services. As a research project, these four programs contain common functionalities that are representative for embedded systems.

By tracing the requirement tag from the testing reports, we collected 3111 test cases for the 4 programs, and identified a total of 128 bugs. Table 2 presents a summary of the frequencies of occurrence for each of the relations that were proposed in Section II.

The software measures collected for this analysis include software size, the dominant bugs’ density, the recessive bugs’ density, the defect density and the three interferences’ density.

<table>
<thead>
<tr>
<th>Name</th>
<th>Lines of code</th>
<th>Programming language</th>
<th>Testcases</th>
<th>Failed Testcases</th>
<th>Bugs</th>
<th>Dominant Bugs</th>
<th>Recessive Bugs</th>
<th>independent-failure</th>
<th>failure-masking</th>
<th>failure-emerging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Driver</td>
<td>35658</td>
<td>C</td>
<td>359</td>
<td>14</td>
<td>19</td>
<td>14</td>
<td>5</td>
<td>11</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Tools</td>
<td>52672</td>
<td>C++</td>
<td>1158</td>
<td>26</td>
<td>30</td>
<td>7</td>
<td>23</td>
<td>20</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Specificlib</td>
<td>49912</td>
<td>C</td>
<td>1353</td>
<td>24</td>
<td>28</td>
<td>12</td>
<td>16</td>
<td>23</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Workspace</td>
<td>19775</td>
<td>C</td>
<td>241</td>
<td>14</td>
<td>51</td>
<td>11</td>
<td>40</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

Detailed descriptions are listed in Table 3, and the descriptive statistics of these samples can be found in Table 4.

B. Empirical investigation

The four research objects are well tested by CATC. We obtained the completed testing reports, including requirement documents, detailed design specification for every module, static analysis reports, unit test reports, integration testing reports and test case specifications. Although the documents for research objects are complete, the required measures for our study is not directly available. Processes are needed to extract relevant data from the reports and calculate the measures that we need.

We needed to link faults to corresponding failures. We tracked the requirement tag to construct the connection. Requirement tag represents the corresponding requirements from client. In design documents, every function has an identified requirement tag. This tag is not unique, because a certain requirement corresponds to multiple functions which are designed to meet this requirement. In the phase of unit test and statics analysis, we can establish one-to-one relationship between reported fault and unit function. After that, in integration testing report, we can get one-to-one relationship between requirement tag and integration test cases. For a certain integration test case, we can track requirement tag to identify the unit function triggered by specified test case, then map the unit function to certain single fault in program. With this tracking framework, we can map failed integration test cases to their corresponding faults.

What should be noted is that there will be version upgrade between unit test and integration test, in order to eliminate this bias, we run the integration test cases at the program version which run the unit test. For a third-party test, there is test specification or called test oracle developed by programmers and managers of tested software. We compared the outputs executed by test cases to the test specification. If the outputs differed, then the execution of the test case was said to have resulted in ‘failure’. If the outputs were the same, then the test case execution was considered to have been ‘successful’. This is consistent with taxonomy in [20] where a failure is defined as an event that occurs when a delivered service deviates from correct service. All program executions were on PC with Testbed 9.4.4, the compiler used was GCC 3.4.3, the operating system was win7 (64 bit) with QNX 6.5 compile environment.
TABLE III. THE SOFTWARE MEASURE

<table>
<thead>
<tr>
<th>Measure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software size</td>
<td>Kilo line of code (KLOC)</td>
</tr>
<tr>
<td>Dominant bugs’ density</td>
<td>(The number of dominant defects)/ KLOC</td>
</tr>
<tr>
<td>Recessive bugs’ density</td>
<td>(The number of recessive defects)/ KLOC</td>
</tr>
<tr>
<td>Total defect density</td>
<td>The total number of defects/KLOC</td>
</tr>
<tr>
<td>The density of interference</td>
<td>(The number of failed testcases that correspond to this interference) / (the number of failed testcases)</td>
</tr>
<tr>
<td>independent-failure</td>
<td></td>
</tr>
<tr>
<td>The density of interference</td>
<td>(The number of failed testcases that correspond to this interference) / (the number of failed testcases)</td>
</tr>
<tr>
<td>failure-masking</td>
<td></td>
</tr>
<tr>
<td>The density of interference</td>
<td>(The number of failed testcases that correspond to this interference) / (the number of failed testcases)</td>
</tr>
<tr>
<td>failure-emerging</td>
<td></td>
</tr>
</tbody>
</table>

TABLE IV. THE DESCRIPTIVE STATISTICS OF SAMPLES

<table>
<thead>
<tr>
<th>Measures</th>
<th>Mean</th>
<th>Minimum</th>
<th>Median</th>
<th>Maximum</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software size</td>
<td>39.504</td>
<td>19.775</td>
<td>42.785</td>
<td>52.672</td>
<td>13.090</td>
</tr>
<tr>
<td>Dominant bugs’ density</td>
<td>0.330</td>
<td>0.133</td>
<td>0.316</td>
<td>0.556</td>
<td>0.184</td>
</tr>
<tr>
<td>Recessive bugs’ density</td>
<td>0.730</td>
<td>0.140</td>
<td>0.378</td>
<td>2.023</td>
<td>0.870</td>
</tr>
<tr>
<td>Total defect density</td>
<td>1.061</td>
<td>0.533</td>
<td>0.565</td>
<td>2.579</td>
<td>1.012</td>
</tr>
<tr>
<td>The density of interference</td>
<td>0.735</td>
<td>0.428</td>
<td>0.778</td>
<td>0.958</td>
<td>0.222</td>
</tr>
<tr>
<td>independent-failure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The density of interference</td>
<td>0.064</td>
<td>0</td>
<td>0.0565</td>
<td>0.143</td>
<td>0.060</td>
</tr>
<tr>
<td>failure-masking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The density of interference</td>
<td>0.200</td>
<td>0</td>
<td>0.151</td>
<td>0.5</td>
<td>0.221</td>
</tr>
<tr>
<td>failure-emerging</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

V. ANALYSIS AND RESULTS

A. RQ1: How often does the independence assumption in a multi-faults program hold?

The independent-failure has the highest frequency of occurrence in failed test cases. The proportion of independent-failures is 76.9% of the total failures. For all the four programs, the mean proportion of independent-failure is 73.5%. This result is consistent with research [12], in which the authors observed that the influence of multi-faults was not as great as expected.

We furtherly traced the requirement tag in every failed test cases to construct the link between faults and failures. For all 78 failed test cases, we traced the triggered faults and the failure manifestation, the result is shown in Figure 1.

![Fig. 1 The triggered faults by failed test cases](image)

The distribution of triggered faults in Figure 1 tells us that in most cases, the failed test cases triggered only one fault in software. The proportion of test cases triggering only one fault has reached to 58% in our study. If we only consider the failed test cases that triggered multi-faults (Figure 2), we still have 70% chance to trigger only two faults in a program at the same time.
B. RQ2: Do interactions between multi-faults cause new failure behaviors?

From Table 2 and Table 4 we can see that the number of test cases that cause new failure manifestation is 14, which is three times of that for the failure-masking interference. This means in our case, the multi-faults interactions are complex and corresponding failures is more often to be unexpected.

We further examined the sources of the new failures, shown in Table 5. Compiler/environment or test tool error is the main cause of new failure manifestation. This type of failures accounts for half of new failure manifestation in our case.

<table>
<thead>
<tr>
<th>Source of new Failures</th>
<th>The number of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler/environment or test tool error</td>
<td>7(50%)</td>
</tr>
<tr>
<td>Integration fault</td>
<td>3(21.4%)</td>
</tr>
<tr>
<td>Requirements p</td>
<td>3(21.4%)</td>
</tr>
<tr>
<td>Data problem</td>
<td>1(7.2%)</td>
</tr>
</tbody>
</table>

As a conclusion, the interactions of multi-faults can cause new failure behaviors. Such kinds of interactions can bring great uncertainty to the system, manifesting themselves as environment error, etc. In spite of the low proportion of occurrence of failure-emerging (5% chance in our investigation), the result of this uncertainty brought great harm to the system for its unpredictable failure behaviors (3 failed test cases of system crash were caused by compiler/environment or test tool error in our case).

C. RQ3: Do the interactions between multi-faults mask or cover some failures' manifestation?

The data of masking failure, shown in Table 2, accounted for only 5.12% of the total failed test cases. Each of these four test cases triggered two or more dominant faults, however, corresponding failure only showed one dominant fault. For example, a memory leak caused by the array bounds covered the computing fault. The small sample size does not allow us to make any further conclusion about the mechanism of failure-masking.

D. RQ4: What is the distribution characteristics of the dominant and recessive faults?

The distribution of dominant and recessive faults in the general application software development platform is shown in Figure 3. As can be observed from the data, the dominant faults accounted for one-third of the total faults, and the recessive faults accounted for another two-third of the total faults correspondingly.

The failure modes of recessive faults and dominant faults are given in Figure 4. The distribution of dominant faults are concentrated on the faults of “inconsistency of requirement and design” and “coding fault”. This is consistent with study [10], which found that the requirements fault and coding fault are two major compositions of failure type.

However, for the recessive faults, the document fault accounted for almost 70%, followed by coding fault, which nearly accounted for the rest 30%. What should be noted is that the fault types of coding fault in dominant faults and recessive faults are quite different in our empirical study. For dominant faults, the coding faults are the data processing faults in system, but for recessive faults, the coding faults are redundant code and dead code.

E. RQ5: What is the effect of dominant and recessive faults on failure behaviors?

In order to investigate the effect of dominant and recessive faults, we searched all 78 failed test cases and constructed the
linkage between failure and its causation. The result is shown in Figure 5.

![Figure 5](image)

Fig. 5 the proportion of causation of failed test cases

As shown in Figure 5, the main cause of failures are the interactions of dominant faults. On the contrary, the recessive faults show no significant effect on the failure manifestation: only one failure is caused by the collective effect of recessive fault and dominant fault. In this failed test case, the dominant fault (data processing error) was triggered and activated the recessive fault which will not be triggered in normal case. This interaction finally caused memory leak, leading to a crash of the system.

Like the uncertainty of failure-emerging, the interaction of dominant fault and recessive fault have very few chance (1% in our case) to manifest itself. In most cases, the dominant faults is the main cause to a failure, and the recessive fault keeps ‘silent’. However, once triggered, a recessive fault could become a severe risk that cannot be ignored.

VI. DISCUSSION

In the research of debugging, fault localization is a popular and efficient technique that help programmers find and fix bugs in finding a good starting point. But the efficiency of fault location techniques such as Tarantula[21], Ochiai[22]. However, these tools suffer from the interactions of multiple faults[5]. For instance, the effectiveness of Tarantula technique declines on all faults as the number of faults increases[21]. Y. Singh stated that multiple faults in a software many times prevent debuggers from efficiently localizing a fault[23]; further, Denmann et al. describes coverage-based FL requiring an implicit assumption that multiple faults be independent of one another, or they will not produce “good results”[24].

Since the side effect of multiple-faults to fault localization technique has been accepted by common research, the distribution and the characteristic of multiple-faults interactions become an interesting topic. However, the researches aimed to this topic are limited. Up to now, representative achievements are [12, 17, 18]. In these researches, the research paradigm is comparing the test cases between single-fault versions and multi-faults versions which are obtained by injecting known bugs manually. They studied the number of faults from 1 to 10 or more and researched the distribution of multiple-faults interactions. However, as stated in previous chapter, due to failure masking and the implementation of software, this research method has some inherent restrictions. According to the result of multiple-faults interaction data in this paper, we can observe that the faults density will not be very high in the third party testing software. For instance, if we only consider the failed test cases that triggered multi-faults, we have 70% chance to trigger only two faults in a program at the same time. In our empirical study, it is not likely to trigger 6 or more faults at one time. Our result is not suggest that the method of injecting faults to analyze the effect of multiple-faults is unsuitable, on the contrary, the result of our empirical study is consisted with [17] in low fault density.

The method of injecting faults can help researcher to analyze the global impact of multiple-faults interactions. But for different stages of software implementation, the distribution and the characteristic of multiple-faults interactions need to be carefully researched.

The interactions of dominant faults and recessive faults are also interesting to be further studied. In our case, most failure behaviors of multi-faults can be accounted by interactions of dominant faults. As expected, the recessive faults are often masked by dominant faults. However, in specific conditions, recessive fault has negligible effect on the failure manifestations in multi-faults programs. In research[12], Jones finds that the presence of the fault reduced the ability of the coverage-based fault localization technique to effectively localize the other faults. The conclusion is similar to our result that the dominant faults have masked the recessive faults. We believe there are connections between failure-masking mechanism and the interactions of dominant faults and recessive faults. We will study that in future study.

VII. CONCLUSION

This paper performed an empirical analysis on the failure behaviors of multi-faults for four industrial software systems. In our case, we find that 73.5% failures are triggered by individual test cases. This implies a high chance to support the independence assumption, which in turn suggests that it is achievable to localize multi-faults by decomposing a set of multi-faults into several single-faults in most cases in our objective software systems. Furthermore, in our case, less than three faults in multi-faults programs are concurrently trigged by one test case with the proportion of 75% in all failed test cases. Finally, the failure behaviors of multi-faults in our case can mainly accounted by interactions of dominant faults. The recessive fault has negligible effect on the failure manifestations in multi-faults programs.

The limitation of this study is that the observed results are limited to the four software systems. Furthermore, the defect data were obtained through independent third-party acceptance tests at the time of delivery. The extent to which the observed phenomenon can be extended to other programs in other development phases requires further studies.

REFERENCES


