An insight of double-faults interactions in program: an empirical study

Jianxing LI, Xiaobo YAN *, Bin LIU *, Shihai WANG
Science & Technology on Reliability & Environment Engineering Laboratory
Beihang University
Beijing, China
2455184342@buaa.edu.cn, yxbbuua@buaa.edu.cn, liubin@buaa.edu.cn, wangshihai@buaa.edu.cn

Abstract—The study of multiple faults is becoming a hot spot. In the large software program, Multiple faults may interact with each other in some ways. In this paper, we investigated the property of multi-faults through the method of fault-injection in Siemens suite. Inspired by previous research, we focused our attention on three types of fault interaction: independent faults; faults masking; faults construction and explored what factors lead their interference in nature. The empirical study in Siemens suits showed that: 1). the probability of two faults interference is less than 1%, which means the independent assumption holds true in most cases. 2). faults masking is more frequent than the fault construction. 3). the occurrence of faults interference is not random. By focusing on and analyzing the double-faults versions that occur high-frequency interference, we find they always involve the same variable, which means the occurrence of fault interference have a certain condition.

Keywords—multi-faults;faults localization; software;destructive interference; constructive interference;

I. INTRODUCTION

Considerable progress has been made in software defect prediction and software bug repair. As the continuous expansion of software size and complexity, traditional software debugging seem to be a different task that requires time and com. As a solution, software fault localization is an actively researched topic in software debugging today.

Wong et al catalog and provide a comprehensive overview of software fault location techniques and discuss key issues and concerns that are pertinent to software fault localization as a whole. They studied 331 papers published from 1977 to November 2014, find that the majority of published papers in software fault localization focus on programs with a single bug [1], researchers tend to assume that each faulty program has exactly one bug. However, the empirical study in [2, 3] implies that individual failures are often triggered by multiple bugs spread throughout the system! This observation raises doubts concerning the validity of some heuristics and assumptions based on the single-fault scenario.

For single fault, we have slice-based [4], spectrum-based [5] methods to locate the fault. For multiple faults, a popular assumption is that different faults in the same program fail independently [6]. A representative theory under this assumption is Bayesian reasoning framework for software fault location [6-8]. However, a potential weakness of these methods is that they all assume program components fail independently; in other words, interferences between multiple faults are ignored, which is not necessarily the case in practice. Further, Jones [9] reported that multiple faults have a negligible effect on the effectiveness of the fault localization.

Thus, we need an overall study in the nature of multi-faults. In that field, Debroy and Jones [10,11] made important attempts to research the multi-faults interaction by comparing the differences of execution results between the single-fault versions and multi-faults versions suggest that no one form of interference is unconditional and the situation of failure masking is more frequent than leading to a new failure. Although their experiments are specific and rigorous, the failed test cases didn’t be traced to the causative fault.

The research of multiple faults is related to the internal structure of the software module, which is closely related to the fault distribution and type of the software fault. Multi-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. Multiple faults may collectively produce failure on a test case that does not fail due to any single fault alone. Thus, we need to analyze the interaction of multi-faults from the perspective of internal attributes of faults such as fault type and structure of program.

Inspired by previous study [10,11], using the Siemens suite [12], the objective of this study is to observe interactions between multi-faults interference and the attributes of faults injected and to empirically deduce some helpful conclusions for fault localization. We do this by first collect each fault attributes in the Siemens suite [12] and the software operation results with single faults and double faults. Then we analysis the relationship between data and attributes graphically. For convenient fault injection and data analysis, we study double faults interference only. Essentially, the conclusion of multiple faults (n-faults) is the same as double faults because we can combine n-1 faults as a single fault.

The remainder of this paper is organized as follows: Section 2 presents the methodology used in research. The use of the method is illustrated by a case study in Section 3. The results of different algorithms are compared with each other in Section 3 as well. Section 4 presents conclusions and suggestions for future work.
II. METHODOLOGY

This section first presents a detailed account of faults interferences in multi-faults programs and provides illustrative examples for fault attributes collection. Additionally, detailed experiment steps are described to subsequent case verification in this paper.

A. The description of faults interference

As for multiple faults, we pay attention to 3 situations: independent faults, faults masking and faults construction. The definition of these interactions are shown as below:

Inspired by definitions in previous study [10], the fault masking is consisted with three cases, we call them the same test case fails due to a single fault F2. The definitions type classification, whose purpose, perspective and complexity test case execution, in order to eliminate the interaction of test cases.

property of fault, including functional fault, interface fault, program are independent and show no obvious interactions. For M2, M3 are similar.

The fault construction only contains one case. Given two faults F1 and F2, when we run a test case t that fails in a single-fault version containing F1, but no longer fails when another fault F2 is added, whereas the same test case fails due to a single fault F2. The definitions of M2, M3 are similar.

The fault construction only contains one case. Given two faults F1 and F2, when we run a test case in the program with faults F1 and F2 successfully but the results are failed both in the single-fault version with either F1 or F2. We call this situation R. If there is no faults masking or fault construction, we call two faults independent, which means the faults in the program are independent and show no obvious interactions. For ease of understanding, Table 1 shows the interpretation of four interference. In Table 1, ‘fail’ and ‘succeed’ mean the results of test case execution, in order to eliminate the interaction of different test cases, all rows in Table 1 was tested with same test cases.

Table I. The interpretation of four interference

<table>
<thead>
<tr>
<th>Interference</th>
<th>Program with single F1</th>
<th>Program with single F2</th>
<th>Program with F1&amp;&amp;F2</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>fail</td>
<td>fail</td>
<td>succeed</td>
</tr>
<tr>
<td>M2</td>
<td>fail</td>
<td>succeed</td>
<td>succeed</td>
</tr>
<tr>
<td>M3</td>
<td>succeed</td>
<td>fail</td>
<td>succeed</td>
</tr>
<tr>
<td>R</td>
<td>succeed</td>
<td>succeed</td>
<td>fail</td>
</tr>
</tbody>
</table>

B. The collection of fault attribute

What factors can influence interferences between multiple faults? In this paper, we suspect there are three faults attributes-fault position, fault type and the variable involved by injected fault. Firstly, fault position is the important fault attribute. In this paper, fault position refers to the function where the fault exists. Secondly, software fault is built on the base of classification of fault type [13]. There are many kinds of fault type classification, whose purpose, perspective and complexity are not identical. This paper adopts Thayer software fault classification method [14], which classified according to the property of fault, including functional fault, interface fault, logic fault, computational fault, data processing fault, etc. Through the classification and statistics, we can understand the influence of faults. Particularly, this classification is applicable to guide developers to detect elimination and software optimization. Finally, what we collect is the variable involved and affected directly by the existing faults. Because the variable is the most direct way to change the result.

The detailed interpretation of attributes and provides an illustrative example for collecting the attribute value in Table 2 and Table 3. In Table 2, there are two different versions of the function Sample. The left column is normal and the right column is two faults. Then we analyze the construction of the program and record the attributes of faults in Table 3 for collecting data in case study.

Table II. An example for collecting the attribute value

<table>
<thead>
<tr>
<th>Correct Program</th>
<th>Program with two faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample( a, b) {</td>
<td>Sample( a, b) {</td>
</tr>
<tr>
<td>test( a, b);</td>
<td>test( a, b);</td>
</tr>
<tr>
<td>s = a;</td>
<td>= a;</td>
</tr>
<tr>
<td>h = b;</td>
<td>b;</td>
</tr>
<tr>
<td>result = 0;</td>
<td>result = 0;</td>
</tr>
<tr>
<td>if( (s + h) &gt;= -7) ; result = 1;</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>result = 2;</td>
<td>result = 2;</td>
</tr>
<tr>
<td>cout&lt;&lt; (result )</td>
<td>cout&lt;&lt; (result )</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Table III. The attributes of above faults

<table>
<thead>
<tr>
<th>Fault</th>
<th>Lines</th>
<th>Position</th>
<th>Type</th>
<th>Variable involved</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>3</td>
<td>Sample</td>
<td>computational</td>
<td>x</td>
</tr>
<tr>
<td>F2</td>
<td>6</td>
<td>Sample</td>
<td>logic</td>
<td>result</td>
</tr>
</tbody>
</table>

In order to reduce the interference of irrelevant variables, and to facilitate data collection, this paper adopts some specific methods. At first, we select some faults with different types of the benchmarks. For each benchmark version, we get the double fault version after injecting another conflict-free fault. Secondly, Control the type of faults injects in double faults version that transforms from the same single fault version. Each version contains one same fault and another diverse fault. Finally, all test case executing environment for all versions is the same. The operating system was Ubuntu (x64) and the compiler is Gcc.

C. Experimental procedure

Generally speaking, this is an exploratory paper, but our experimental design is distinct and rigorous. Based on the above ideas, the steps of study case are as follows:

1. Get access to fault-free software and single fault version, correspondingly.
2. Select one of the single fault software as the benchmark. We will inject an extra fault from another single fault version to get a double fault version.
3. Repeat step2, get a number of double fault version software.
4. Collect attribute value of faults injected, including the position of the fault, variables involved and the type of fault.
5. The single fault software version and the double fault software version will be executed through the same test cases, record the results whether a failure occurs.
6. Count the frequency of each situation.
7. Analyze the relationship between fault attributes and interference situation.

III. CASE STUDY

In order to empirically analyze how interference mentioned in Section 2 occurs on actual programs, we performed a case study in the Siemens suite [12]. This section describes the subject programs, the data collection and the conclusion analysis.

The seven programs in the Siemens suite have been well studied and used for several fault localization studies [15, 16, 17], as an epitome choice. We can download the normal and faulty versions as well as the respective test cases from [12]. In this paper, in order to make the experiment more tractable, only part of programs are used, such as tcas and print_tokens. Recall in Section 2, when considering double-fault version program, we need to collect each fault attribute values used and seed different faults into the single-fault program. So, before injection, it is necessary to understand the structure of the program in order to eliminate conflicts between two faults used. Because some conflicting faults local in the same location and thus cannot be seeded, simultaneously. Taking tcas as an example, the structure of the main function Alt_sep_test is given in Fig 1. Alt_sep_test calls four remaining functions, each of which returns a value through a series of quite simple computation. The 13 global variables are initialized through command line argument as the test case. Most functions execute the logical operation. When two conflicting faults share the same location and cannot be seeded simultaneously, we discarded this combination. In this paper, we choose seven faults as a benchmark, then seed remaining faults into them, respectively, except conflicting faults. To make experiment more representative and extensive, the benchmark faults chosen in this paper distribute in different functions, involve different variables and have different types as far as possible.

A. Data Collection

In order to eliminate research bias, we adopt all non-conflicting faults and all 1608 test case. In tcas, there are 41 versions with a single fault. The objective of this experiment is analyzing the relationship between interference and faults.

```
enabled, tcas_equipped, intent_not_known Declaration

enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
tcas_equipped = Other_Capability = TCAS_TA;
intent_not_known = Two_of_Three_Reports_Valid && Other_RAC = NO_INTENT;

Enabled=1
( (tcas_equipped && intent_not_known) || !tcas_equipped)=1

need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();

need_upward_RA

need_downward_RA

alt_sep = UPWARD_RA
alt_sep = UNRESOLVED
alt_sep = DOWNWARD_RA
alt_sep = UNRESOLVED

Figure 1. The control flow of the main function Alt_sep_test
```
attributes, some of which are similar. So, understanding and distinguish each fault’s attribute is useful for subsequent analysis. What we extract here are fault type, involved variables, and the position they distribute. The extracting rules is mentioned is mentioned in section 2. The benchmark fault attributes are presented in Table 4. In “Function” column, each number represent a different function. Number ‘1’ is the function “Non_Crossing_Biased_Climb ( )”. Number ‘2’ is the function “Non_Crossing_Biased_Descend ( )”. Number ‘3’ is the function “Own_Below_Threat ( )”. Number ‘4’ is the function “Own_Above_Threat ( )”. Number ‘5’ is the function “Alt_sep_test ( )”. And “Macro” and “variable” represent the faults locate in variable definition and macro definition.

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 [18] where a failure is defined as an event that occurs when a delivered service deviates from correct service.

B. Result

In this section, we draw some conclusions. Table 4 presents the summary of some interference with high frequency and Fig 2 show it graphically. Please pay attention that the data presented corresponds to the interferences between double faults which we generate based on each benchmark fault, where the benchmark fault is from single fault version 1, 5, 14, 23, 31, 33 and 38 respectively. Viewed as a whole, the data collected is indicative that not all interferences hold uniformly throughout. The independence assumption holds in most of the time (more than 99%). This means that only about 1% of the time we can see the faults interferences. We observed that failure masking is more frequent than faulty construction. The proportion of fault masking is 0.51% and fault construction only 0.13%. Some double-fault versions even do not occur fault construction.

As the benchmark fault change, the interference between two faults is different. When benchmark fault refers to F1 or F3, the probability of fault masking and construction is higher than average value. When the benchmark fault is F23, there is no another fault can mask its influence. How to explain this phenomenon?

From the perspective of each double-faults version, the distribution of each interference is not uniformity and random. When the benchmark fault is F1 and injected version is F2, there are 132 test cases which can mask failure deduced by F1, accounting for 25.7% of all M2. F33 accounts for 25.6% as well. And most of the injected faults cannot mask its failure. The similar situations exist in other double-fault versions. Table 5 presents the noteworthy situations. Each “fault: frequency” presents a situation. For example, the second line “F2: 132” means that the frequency of M2 is 132 when injected faults are F1 and F2. The above analysis firms that the occurrence of interference is concerned with the fault attributes.

At first, we explore the relationship between the fault position and the fault interference. According to the position of each fault we collected in Table 6, we find when two faults are in the same function or involve the same variable, it is more likely to trigger a failure, such as F1 and F40. They are both in function Non_Crossing_Biased_Clim and involve the same variables need_upward_RA. Fault combination (F33, F37) have the same phenomenon as well.

As for the fault masking, we focus on faults (F1, F38), (F1, F2), (F1, F33), (F31, F40), (F14, F33), (F5, F14) and (F14, F27). Our statistics shows that all double-fault combination affect the same variable directly. F1 affects the return value of the function “Non_Crossing_Biased_Clim()”, by removing “&&” from “Down_Separation >= ALIM()”. F33 and F38 affect the return value of “ALIM()”). Both, F14 change the value of Macro definition MAXALTDIFF. F5 remove “Cur_Vertical_Sep > MAXALTDIFF” from the judgment statement “enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep >

![Table V. The frequency of interferences](image-url)
MAXALTDIFF). F5 eliminates the effect of the fault F14. (F14, F22), (F14, F25) and (F14, F27) have the similar explanation.

![Image](image_url)

**Figure 2.** The frequency of interferences

When it comes to the fault type, there is no obvious conclusion. Because the fault combinations which we pay attention to above, do not have coincident fault type combinations. From the results, any combination of the fault type is likely to cause failure masking. F2 and F33 have different fault types but they mask the program failure induced by F1 both. F1 is a logic fault and F37 is data processing fault, and they also mask the failure deduced by F38. F5, F12, F15 and F27 are logic faults and they mask the failure deduced by F14. So, the specific fault type is not a necessary condition to mask previous failures or construct a new failure.

### Table VI. The noteworthy situations of faults interference

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Situations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M3: F3:45</td>
</tr>
<tr>
<td></td>
<td>R: F4:149</td>
</tr>
<tr>
<td>F2</td>
<td>M3: F1:60, F14:50, F30:11</td>
</tr>
<tr>
<td></td>
<td>R: F1:6, F1:5</td>
</tr>
<tr>
<td></td>
<td>M3: F1:48, F16:34, F17:12, F2:12</td>
</tr>
<tr>
<td>F14</td>
<td>M3: F1:44, F12:12, F34:23, F38:10</td>
</tr>
<tr>
<td>F31</td>
<td>M2: F3:14</td>
</tr>
<tr>
<td></td>
<td>M3: F4:125</td>
</tr>
<tr>
<td>F33</td>
<td>M2: F1:85, F14:37</td>
</tr>
<tr>
<td></td>
<td>R: F2:46, F3:118, F4:46</td>
</tr>
<tr>
<td>F38</td>
<td>M2: F1:45, F16:29</td>
</tr>
<tr>
<td></td>
<td>M3: F1:11, F14:22, F16:19, F38:23</td>
</tr>
<tr>
<td></td>
<td>R: F3:7, F4:17</td>
</tr>
</tbody>
</table>

### IV. Conclusion

This paper investigates the interactions that take place between double faults and how these interactions may construct or mask failure. We do so, by making assumptions to some fault attributes that may be related to this interference. And the set of test cases executed in double-faults version and corresponding single-fault versions. Results are suggestive of the fact that the probability of two faults interference is very small, less than 1%. In this 1%, the fault masking is more frequent than the fault construction. The occurrence of faults interference is not random. By analyzing the double-fault versions that occur high-frequency interference, we find they always involve the same variable. The fault function and fault type are irrelevant factors. This conclusion is reasonable. If the two faults involve the same variable, they can change the value of the variable directly and offset the influence each other.

This conclusion is useful in the multi-fault location. In the area of fault location, the technology of Clustering Algorithm is widely used, the phenomenon in this paper indicates that we can use classification algorithm such as Clustering Algorithm to classify the slices of programs which contain the same variable to investigate the effect of multi-fault and build more efficient fault-location method!

Though the results presented are based on the program used widely and the experiment is rigorous, the small sample is the threat to validity. Future work includes is improving our analysis sample and analyzing fault interference on different programs to get more convincing results.
REFERENCES


totle/Tools/subjects, January 2007


