Is MFS Strongly Correlated with faulty code? *

Xintao Niu, Changhai Nie
State Key Laboratory for Novel Software Technology
Nanjing University
China, 210023

Laleh Sh. Ghandehari, Jeff Lei
Department of Computer Science and Engineering
The University of Texas at Arlington

Xiaoyin Wang
Department of Computer Science
The University of Texas at San Antonio

ABSTRACT

Combinatorial Testing (CT) is an effective technique for testing the interactions of factors in the Software Under Test(SUT). Most works in CT focus on the method itself, e.g., how to generate test cases, model the inputs, or handle the constraints of the inputs. Few of the works consider the justification of CT, i.e., is detecting and identifying the failure-inducing interactions really useful and helpful to code-level fault diagnosis? In this paper, we novelty studied the relationship between the failure-inducing interactions and code which causes the failure. Specifically, based on symbolic execution, we firstly obtain the guaranteed code of the corresponding failure-inducing interactions, i.e., those program entities which are directly affected by these interactions. And then we will compared these guaranteed code with those real faulty code to see whether there exists any associations between failure-inducing interactions with these real faulty code. Our empirical studies based on 5 real subjects showed that the failure-inducing interactions are strongly correlated with faulty code in the SUT.

CCS Concepts

•Software defect analysis → Software testing and debugging;

Keywords

Software Testing, Combinatorial Testing, Fault localization, Failure-inducing interactions, Guaranteed code

1. INTRODUCTION

Modern software is becoming more and more complex. To test such software is challenging, as the candidate factors that can influence the system's behaviour, e.g., configuration options, system inputs, message events, are enormous. Even worse, the interactions between these factors can also crash the system, e.g., the incompatibility problems. In consideration of the scale of the industrial software, to test all the possible interactions of all the factors (we call them the interaction space) is not feasible, and even if it is possible, it is not wise to test all the interactions because most of them do not provide any useful information.

Many empirical studies show that, in real software systems, the effective interaction space, i.e., targeting fault detection, makes up only a small proportion of the overall interaction space [14, 15]. What's more, the number of factors involved in these effective interactions is relatively small, of which 4 to 6 is usually the upper bounds[14]. With this observation, applying Combinatorial testing(CT) in practice is appealing, as it is proven to be effective to detect the interaction faults in the system.

CT tests software with a elaborate test suite which checks all the required parameter value combinations, and after detecting some failures by this test suite, it then identify the failure-inducing interactions, or more formally, failure-causing schemas (MFS) in the SUT. Most works in CT focus on the method itself, e.g, to design smaller test suite with the same interaction coverage [3, 4, 16, 12], or to identify the MFS more accurately [20, 22, 35, 24]. Few of the works consider the following question:

Is detecting and identifying the MFS really useful and helpful to code-level fault diagnosis?

To analyse this question is important and necessary, because it will build the relationship between MFS and faulty code, which is the foundation to apply CT on code-level debugging. In this paper, we try to answer this question by studying the *guaranteed code* of interactions. The *guaranteed code* of a interaction is the program entities (e.g., statements, branches, blocks, etc.) which are directly *affected* by the interaction according to the previous study [25]. Obtaining the guaranteed code of an interaction can help us understand how this interaction influence on the behaviour of the program under test. Furthermore, analysing the guaranteed code of the MFS can offer us an insight into the extent to which the MFS is related to the cause of the failure; based on which, we can learn whether detecting and identifying the MFS can facilitate the debugging, as well as bug fixing.

There are many techniques to compute the MFS in CT. In this paper, we adopts our previous method proposed in [22], which is one of the most common MFS identification technique in CT. With respect to guaranteed code, we follow the steps which are original proposed in [25], which firstly

*(Produces the permission block, and copyright information). For use with SIG-ALTERNATE.CLS. Supported by ACM.

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

utilizes symbolic execution tool to search possible paths for different values assigned to the input parameters, and then calculates the guaranteed code for each possible interaction based on these paths. One difference from study in [25] is that we only need to compute the guaranteed code for the MFS we identified in the SUT, instead of all the possible interactions. After obtaining the MFS and corresponding guaranteed codes, we will evaluate the correlation between MFS and faulty code.

We have design several empirical studies on 5 open-source software subjects. These studies considers several different aspects (e.g., the degree of MFS, the types of faults) of the relationships between MFS and faulty code. Our results suggests that: 1) The MFS does relate to the faulty code to some extent; 2) For different types of faults, the correlation between MFS and faulty code varies; 3) The input model of the program under test significantly impacts on this correlation.

The remaindering of this paper are organised as follows: Section 2 gives the preliminaries about Combinatorial testing (especially MFS-related), and basic definitions about Guaranteed code. Section 3 proposes three research questions that needs to be handled in this paper. Section 4 introduces the subjects on which our experiments are conducted on. Section 5 shows the results as well as the analysis. Section 6 discusses the related works. Section 7 concludes this paper.

2. PRELIMINARY AND FORMAL MODEL

This section presents some formal descriptions about MFS and guaranteed code.

2.1 Basic definitions about CT

Assume that the Software Under Test (SUT) is influenced by n parameters, and each parameter p_i can take the values from the finite set V_i, $|V_i| = a_i$ (i = 1,2,..n). We will give some basic definitions which are related to failure-inducing interactions in CT.

Definition 1. A *test case* of the SUT is a tuple of n values, one for each parameter of the SUT. It is denoted as (v_1, v_2,...,v_n), where $v_1 \in V_1$, $v_2 \in V_2$ ... $v_n \in V_n$.

In practice, these parameters in the test case can represent many factors, such as input variables, run-time options, building options or various combination of them. We need to execute the SUT with these test cases to ensure the correctness of the behaviour of the SUT.

Definition 2. For the SUT, the n-tuple (-,v_{x_1},...,v_{x_k},...)is called a k-degree *schema* ($0 < k \leq n$) when some k parameters have fixed values and other irrelevant parameters are represented as "-".

For example, the tuple (-, 4, 4, -) is a 2-degree schema. In effect a test case itself is a k-degree *schema*, when k = n. Furthermore, if a test case contains a *schema*, i.e., every fixed value in the schema is in this test case, we say this test case *contains* the *schema*.

Note that the schema is a formal description of the interaction between parameter values we discussed before.

Definition 3. Let c_l be a l-degree schema, c_m be an m-degree schema in SUT and $l < m$. If all the fixed parameter values in c_l are also in c_m, then c_m *subsumes* c_l. In this case we can also say that c_l is a *sub-schema* of c_m and c_m is a *super-schema* of c_l, which can be denoted as $c_l \prec c_m$.

For example, the 2-degree schema (-, 4, 4, -) is a sub-schema of the 3-degree schema (-, 4, 4, 5), that is, (-, 4, 4, -) $\prec$ (-, 4, 4, 5).

Definition 4. If all test cases that contain a schema, say c, trigger a particular fault, say F, then we call this schema c the *faulty schema* for F. Additionally, if none of sub-schema of c is the *faulty schema* for F, we then call the schema c the *minimal failure-causing schema (MFS)* [22] for F.

Note that MFS is identical to the failure-inducing interaction discussed previously. In this paper, the terms *failure-inducing interactions* and *MFS* are used interchangeably.

2.2 Identification of MFS

When a test case fails during test case, we are still far from figuring out the MFS [5, 19, 20], as we do not know exactly which schemas in the failed test cases should be responsible for the failure. For example, if test case (0, 0, 0, 0) failed during testing, there are six 2-degree candidate failure-inducing schemas, which are (0, 0, -, -), (0, -, 0, -), (0, -, -, 0) , (-, 0, 0, -), (-, 0, -, 0), (-, -, 0, 0), respectively. Without additional information, it is difficult to figure out the specific schemas in this suspicious set that caused the failure. Considering that the failure can be triggered by schemas with other degrees, e.g., (0, -, -, -) or (0, 0, 0, -), the problem of MFS identification becomes more complicated.

In fact, for a failing test case $(v_1, v_2, ..., v_n)$, there can be at most $2^n - 1$ possible schemas for the MFS. Hence, more test cases should be generated to identify the MFS. Next, we will show how MFS identification processes with an example.

Consider an program like Fig 1, which contains four integer parameters: a, b, c, and d. With respect to applying combinatorial testing, we need to first build a input model for this program. For simplicity, let a, b, c and d can take on the following values: $v_a = \{1, 2\}$, $v_b = \{0, 1\}$, $v_c = \{1, 2\}$, and $v_d = \{0, 1\}$, respectively. Assume that through testing of this program, we find a test case, e.g., (a = 1, b = 0, c = 1, d = 1), will trigger an arithmetic exception. Then we will describe how CT identify the MFS in this test case.

A typical MFS identification process is shown in Table 1. In this table, test case t represents the failing test case we aforementioned. To identify the MFS, we mutate one factor of t one time to generate new test cases: $t_1 - t_4$. It turns out that test case t_1 passed, which indicates that this test case break the MFS in the original test case t. So (1, -, -, -) should be a failure-causing factor. Similarly, we can also conclude that (-, -, 1, -) is another failure-inducing factor because of the pass of t_3. Considering that all the other test cases failed, which means no other failure-inducing factors were broken, therefore, the MFS in t is (1, -, 1, -).

This identification process mutate one factor of the original test case at a time to generate extra test cases. Then according to the outcome of the test cases execution result, it will identify the MFS of the original failing test cases. It is called the OFOT method [22], which is a well-known MFS identification method in CT. In this paper, we will focus on this identification method.

2.3 Formal description about guaranteed code

```
1  public float foo(int a, int b, int c, int d){
2    int x, y, z;
3    x = 4;
4    if(a < 2){
5      y = 3;
6      if(c < 2){
7              z =  Math.sqrt(y - x);
8              return z / y;
9      }
10     else
11         return x + y;
12   }
13   else{
14     y = 2;
15     if(b < 1){
16         if(d < 1)
17             return y / (x + y);
18         else
19             return x / (x + y);
20     }
21     else
22         return y * x;
23   }
24 }
```

Figure 1: A simple program *foo* with four input parameters

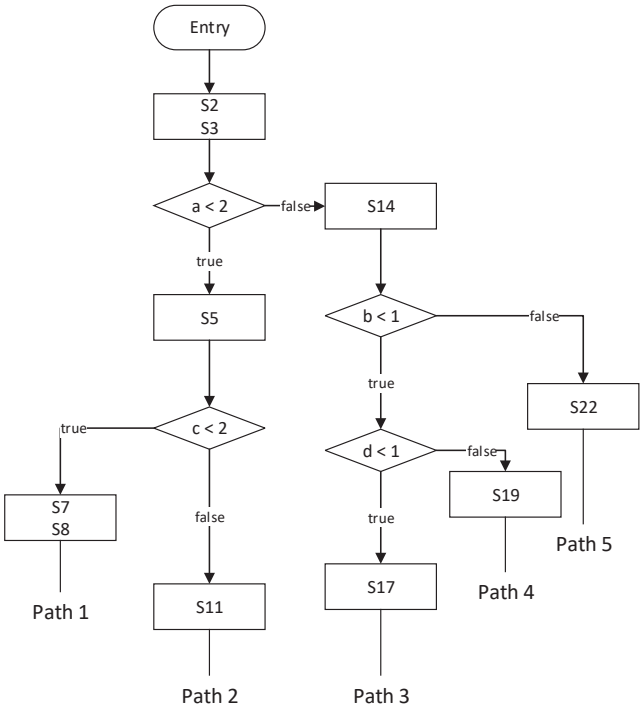

Figure 2: The execution tree of program *foo*

Table 1: OFOT example

	Original test case				Outcome
t	1	0	1	1	Fail
Additional test cases					
t_1	2	0	1	1	Pass
t_2	1	1	1	1	Fail
t_3	1	0	2	1	Pass
t_4	1	0	1	0	Fail

Table 2: Paths, covered entities, and conditions of *foo*

ID	Covered Entities	Path Conditions
1	S2, S3, S5, S7, S8	$a < 2$, $c < 2$
2	S2, S3, S5, S11	$a < 2$, $\neg(c < 2)$
3	S2, S3, S14, S17	$\neg(a < 2)$, $b < 1$, $d < 1$
4	S2, S3, S14, S19	$\neg(a < 2)$, $b < 1$, $\neg(d < 1)$
5	S2, S3, S14, S22	$\neg(a < 2)$, $\neg(b < 1)$

Although MFS are the failure-inducing parts of the failing test input for the SUT, however, we still cannot directly utilize it for fault localization, because they do not provide any code-level information. For this, we need to build the relationship between input schemas and program entities.

Let $\mathcal{P}$ be a path in the SUT. Then let $Cov(\mathcal{P}) = < s_1, s_2, s_3, ...s_n >$ be the program entities that covered by path $\mathcal{P}$. A program entity can be a statement, block, edges, etc. In this paper, we mainly focus on statements; note that other types of structure coverage can also be applied [29]. Let $Pcon(\mathcal{P}) = < pc_1, pc_2, pc_3, ...pc_k >$ be the path conditions that are encountered by path $\mathcal{P}$. As an example, consider the program list in Fig 1. It is easy to find that it has five possible paths, which forms the execution tree in Fig 2.

In this tree, a rhombus node represents a path condition, and a rectangle represents the statement. Note that those consecutive statements are included in one rectangle. From this figure, we can list the paths, their covered entities, and their path conditions, which are explicitly shown in Table 2.

Next we will give some important definitions that are related to the guaranteed code.

Definition 5. For a schema c, and a path $\mathcal{P}$, if all the path conditions in this path can be satisfied when given an input that contains this schema, we call $\mathcal{P}$ the *Consistent path* of schema c, which can be denoted as $SAT(Pcon(\mathcal{P}) \bigwedge c)$.

In fact, to judge whether a path is the *consistent path* of one schema, is to find whether exist an input that contains this schema and follows all the path conditions of this path. Taking the program *foo* for example, Path 1 is the consistent path of schema (1, -, -, -), because $(a < 2 \wedge c < 2 \wedge a = 1) = true$ can be solved, e.g., input (1, 1, 1, 1) is such a result.

Based on this definition, we use the notation $c^\sharp$ to represent the set of all the consistent paths of schema c. Then we will give the formal definition of weak guaranteed of one schema.

Definition 6. For a schema c, the weak guaranteed code of c, i.e., $wg(c)$, is the intersection of program entities covered by its consistent paths. Formally, $wg(c) = \bigcap_{\mathcal{P} \in c^\sharp} Cov(\mathcal{P})$.

To better illustrate the weak guaranteed code of schemas, we list some schemas in the program *foo*, their consistent paths and weak guaranteed code in Table 3. For example, for schema (1, -, -, -), its consistent paths are Path 1 and Path 2, because we can find inputs that contain this schema

and satisfy all the path conditions of them. For example, (1, 1, 1, 1) can go through Path 1 and (1, 1, 2, 1) can go through Path 2. These two inputs both contain the schema of (1, -, -, -). Hence, the weak guaranteed code of $(1, -, -, -)$ are the intersection of the entities covered by these two paths, i.e., S2, S3, and S5.

Note that there is one special schema (-, -, -, -), listed in this Table, it is the sub-schemas of all the other schemas of program *foo*. What's more, it is consistent with all the paths of program *foo*, and hence its weak guaranteed code are S2 and S3, which are the common code of all the paths.

Table 3: Weak guaranteed code of some schemas of *foo*

Schema	$c^\sharp$	$wg(c)$
$(1, -, -, -)$	Path 1, 2	S2, S3, S5
$(1, -, 1, -)$	Path 1	S2, S3, S5, S7, S8
$(1, -, 2, -)$	Path 2	S2, S3, S5, S11
$(2, -, -, -)$	Path 3, 4, 5	S2, S3, S14
$(2, 0, -, -)$	Path 3, 4	S2, S3, S14
$(2, 0, -, 1)$	Path 4	S2, S3, S14, S19
$(2, 1, -, -)$	Path 5	S2, S3, S14, S22
$(-, -, 1, -)$	Path 1, 3, 4, 5	S2, S3
$(-, -, -, -)$	Path 1, 2, 3, 4, 5	S2, S3

Above all, we can observe that, the weak guaranteed code[1] of one schema, essentially, is the maximal common code that is expected to be executed by any path which consists with this schema. In other word, this schema guarantees some code to be executed.

Although the weak guaranteed code always follows with the corresponding schema, it does not necessarily indicates that the schema directly controls the weak guaranteed code. This is because, for one schema, there may exists some other schemas that may always appears with this schema. Hence, it may result in that some code are the weak guaranteed code of one schema, but these code may be actually controlled by other schemas. It is easy to learn that these "accompanying" schemas are the sub-schemas of one schema.

For example, with respect to program *foo*, schema (1, -, 1, -) always appears with schema (1, -, -, -), hence, the weak guaranteed code of schema (1, -, 1, -), i.e., (S2, S3, S5, S7, S8) must always contain the weak guaranteed code of schema (1, -, -, -), i.e., (S2, S3, S5). More formally, we can conclude the relationship of the weak guaranteed code between subsuming schemas as the following proposition.

PROPOSITION 1. *Given schemas s_1, s_2, where $s_1 \prec s_2$, then $wg(s_1) \subseteq wg(s2)$.*

It is easy to prove this proposition and we will omit it. Based on this proposition, to understand which code are under the direct control of some schemas, we need to remove the influence from their subschemas.

Definition 7. For a schema c, the strong guaranteed code of c, i.e., $sg(c)$, is the weak guaranteed code of c with removing those weak guaranteed code of its subschemas. Formally, $sg(c) = wg(c) \backslash \{\bigcup_{c' \prec c} wg(c')\}$.

[1]The definition of weak guaranteed code is similar to the *guarantee coverage* defined in [25]. The difference is that in this paper we do not distinguish the input and value symbolic; instead, they are handled in an unified way.

Strong guaranteed code are the program entities that under the direct control of the corresponding schema, and hence it reflects how the schema influence on the behaviour of program. As an example, considering the schemas which we show their weak guaranteed code in Table 3. We list the weak guaranteed code of all their subschemas, and the strong guaranteed code of themselves in Table 4. For example, for schema (1, -, 1, -), its weak guaranteed code are (S2, S3, S5, S7, S8), while the weak guaranteed code of all its subschemas are: (S2, S3, S5), (S2, S3), (S2, S3) for schemas (1, -, -, -), (-, -, 1, -) and (-, -, -, -), respectively. Hence, the union of the weak guaranteed code of its subschemas are (S2, S3, S5), and the strong guaranteed code of (1, -, 1, -) are (S7, S8).

Table 4: Strong guaranteed code of some schemas of *foo*

Schema	*wg of subschemas*	$sg(c)$
$(1, -, -, -)$	S2, S3	S5
$(1, -, 1, -)$	S2, S3, S5	S7, S8
$(1, -, 2, -)$	S2, S3, S5	S11
$(2, -, -, -)$	S2, S3	S14
$(2, 0, -, -)$	S2, S3, S14	-
$(2, 0, -, 1)$	S2, S3, S14	S19
$(2, 1, -, -)$	S2, S3, S14	S22
$(-, -, 1, -)$	S2, S3	-
$(-, -, -, -)$	-	S2, S3

In this paper, we focus on the guaranteed code of MFS, instead of all the other schemas in the test case. With respect to the example in Fig 1, it is easily to find that the weak guaranteed code and strong guaranteed code of the MFS (1, -, 1, -) are, (S2, S3, S5, S7, S8), and (S7, S8), respectively. They are correlated to the faulty code, i.e., S7, where an exception of taking square root of a negative number will be triggered. This example, especially for the strong guaranteed code, implies that MFS is closely related to the faulty code. However, in the real situation, we do not know whether the conclusion can still hold. As we want to know whether to detect and identify the MFS is really helpful in the code-based diagnosis, hence, we need to do more empirical studies to verify the conclusion.

3. RESEARCH QUESTIONS

To comprehensive study the correlativeness between the MFS and faulty code, we propose three research questions in the following.

3.1 The correlation between MFS and faulty code

As discussed before, it is important to study the relationship between MFS and faulty code. Although the simple example in Section 2.3 shows there exists a strong correlation between the guaranteed code , it is necessary to verify it on more real program subjects. Hence, it motivates our first research question, which is also the key reteach question, that is :

Q1: Is the guaranteed code of the MFS correlated to the faulty code in real program subjects?

To answer this question, we need to conduct studies on a batch of program subjects. In these studies, the MFS,

their weak and strong guaranteed code, and the fault code should be investigated and analyzed. Another point that needs to note is how we evaluate the correlation between the guaranteed code and faulty code. Inspired by the ranking formula that is used in fault localization [21, 1], in this paper, we adopts the following formula to compute the relevance between two codes, i.e.,

$$Correlation(A,B) = \frac{Common(A,B)}{Common(A,B) + Different(A,B)}. \quad (1)$$

Here, $Common(A,B)$ means the number of common statements between two code blocks A and B, and $Different(A,B)$ means the number of different statements contained in them. Based on this formula, more number of common code indicates a closer correlation, while more number of different code, on the contrary, indicates a more distinct or irrelative relationship.

3.2 The influence of different types of faults

According to the nature of the defect, e.g., missing construct, or wrong construct, faults can be categorized into many types. The influence of different type of fault varies widely; as a consequence, the MFS and corresponding guaranteed code of different type of fault may also varies. Hence, focusing on single type of fault may significantly impact on the generality of our study about the correlation between MFS and faulty code, which motivates the second research question:

Q2: To which extent does different type of fault influence on the correlation between MFS and faulty code ?

To answer this question, we need first give the the characterization and classification of software faults. According to the study [7], we decide to conduct experiments on the fault types listed in Table 5. These faults are classified according to the three ODC [2] classes, i.e., Assignment, Checking, and Interface faults. Note that we have omit two more types of faults which are originally listed in [7], as those two types of faults are related to the design or requirement faults. For each of these faults, they are refined into three sub-types, which are based on the nature of the defects, i.e., the fault caused by missing construct, wrong construct, or superfluous part. The column "Example" in Table 5 shows some samples of the corresponding type of fault.

Based on the categories given in Table 5, we next study these faults and their corresponding MFS to observe whether there exists any distinction among different types.

3.3 The influence of inputs model

The last research question is raised from CT itself. It is known that inputs modeling, as the key part of CT [23], significantly influence on the result of CT. For example, if we use the following inputs modeling : $v_a = \{0,1\}$, $v_b = \{0,1\}$, $v_c = \{1,2\}$, and $v_d = \{0,1\}$, respectively, for the program *foo* in Fig 1 instead of what is originally used in Section 2. Then it is easy to compute that MFS of *foo* is (-, -, 1, -), which is different from the original MFS (1, -, 1, -). We can learn the new MFS is irrelevant to the parameter a; this is because according to the execution tree in Fig 2, no matter what a is assigned to (0 or 1), it can only follow $Path1$ or $Path2$, and hence cannot execute other paths. As a result, the fault is only depended on what c is assigned to. The changes of MFS also impacts on the guaranteed code. From Table 3 and 4, the weak guaranteed code of (-, -, 1, -) is $S2$ and $S3$, while there is no strong guaranteed code. This results shows that the correlation between MFS and the real faulty code ($S7$) is very trivial, which is different from the conclusion which is based on the original inputs modeling.

Considering this, the third research question is:

Q3: How does the inputs model of CT affect the correlation between MFS and faulty code?

To answer question, for each program subject in our empirical studies, we design 3 different inputs model (they vary in the the number of parameters and the number of values for each parameter). Then we will obtain their MFS, guaranteed code, and the correlation between MFS and faulty code, respectively. At last, we will evaluate whether there exist any difference among these results.

4. SUBJECT PROGRAMS

We have prepared 8 program subjects for our experiment, of which 7 subjects programs belong to the Siemens set [11], and the remaining subject is GNU Grep[2]. We obtain these subjects, as well as their program specifications, from the Software Infrastructure Repository (SIR) [6]. These subjects have been wildly used in the studies of fault detection and localization [33, 13, 26, 8]. Table 6 shows the specific program subjects, number of lines of code, the brief introduction, and the number faulty versions for each subject.

4.1 Fault characteristics

For each program subject in our experiment, there are several faulty versions come with the correct version. These faults vary in different types and locations. With respect to the research question 2, we need to classify these faults into 3 main types and each of them has 3-sub types. For this, we checked the source file for each version of the specific programs, compared them with the correct version, and then classified the faults according to the aforementioned types. Table 7 shows the fault type distribution for each subject.

Based on this table, we can learn that.

4.2 Input modeling

To test each subject program, we need to model their inputs space firstly. Specifically, We initially followed the instructions described in [9] to build the corresponding model for each subject, which include defining the key parameters and the values of each parameter for the programs, obtaining the possible constraints amon these parameters. For example, for the program *replace* in the Siemens suite, we may consider there parameters: pattern, substitute and input text. Each of them have different type of value, e.g., the pattern may be a common character, digit, or line matching signal, etc. Besides this initial model, we built 2 more inputs modeling based on the initial one (With removing some parameters or values), such that we can observe and compare the results with different modeling. Table 8 listed these models (where *Model1* is the initial model obtaining from [9]) and the number of constraints of each subject. The model is presented in the abbreviated form $\#values^{\#number\ of\ parameters} \times ....$

5. RESULTS

[2]https://www.gnu.org/software/grep/

Table 5: Fault types according to nature and ODC class

ODC class	Nature	Example
Assignment	missing	A variable was not assigned a value, a variable was not initialized, etc.
	wrong	A wrong value (or expression result, etc) was assigned to a variable
	extraneous	A variable should not have been subject of an assignment
Checking	missing	An "if" construct is missing, part of a logical condition is missing, etc.
	wrong	Wrong logical expression used in a condition in branch and loop construct.
	extraneous	An "if" construct is superfluous and should not be present
Interface	missing	A parameter in a function call was missing; incomplete expression as used as parameter
	wrong	Wrong information was passed to a function call (value, expression result, etc.)
	extraneous	Surplus data is passed to a function

Table 6: Characteristics of subject programs

Subject	Loc	Description	# of faulty versions
printtokens	472	lexical analyzers	7
printtokens2	399	-	10
replace	512	performs pattern matching and substitution	32
schedule	292	priority schedulers	9
schedule2	301	-	10
tcas	141	aircraft collision avoidance system	41
totinfo	440	computes statistics given input data	23
grep	10068	searching for lines matching a regular expression.	5

Table 7: Fault type distribution of subject programs

Subjects	Assignment			Checking			Interface		
	missing	wrong	extraneous	missing	wrong	extraneous	missing	wrong	extraneous
printtokens									
printtokens2									
replace									
schedule									
schedule2									
tcas									
totinfo									
grep									

Table 8: Inputs modeling of subject programs

Subject	Model1	Model2	Model3	Constraints
printtokens	$2^1 \times 3^1 \times 4^4 \times 5^1 \times 10^1 \times 13^2$			8
printtokens2	$2^1 \times 3^1 \times 4^4 \times 5^1 \times 10^1 \times 13^2$			8
replace	$2^4 \times 4^{16}$			36
schedule	$2^1 \times 3^8 \times 8^2$			0
schedule2	$2^1 \times 3^8 \times 8^2$			0
tcas	$2^7 \times 3^2 \times 4^1 \times 10^2$			0
totinfo	$3^3 \times 5^2 \times 6^1$			0
grep	$2^7 \times 4^1 \times 6^5 \times 9^1 \times 13^1$			2

5.1 General Correlation

5.2 Fault types

We can firstly learn that the correlativeness has no means to do with odc class. only with types.

the faulty code is, wrong the code, extra the code, missing two codes nearest it.

The conclusion is, that the wrong code and extra code are more close, because those which control this is more close to introducing faulty.

While the missing is very. This is because some code may be .

5.3 Inputs modeling

Different modeling,

the coverage (line coverage, condition coverage), correlated to the faulty code with guaranteed code distance.

(Note that in some pictures, the condition coverage cannot be 100 percent because some exception will not be triggered in the current inputs, e.g., the input memory exception handle module will not be triggered)

Above all, the answer to the question 3 is :

The more the coverage the inputs modeling supports, the more accurate the MFS is correlated to the faulty code.

5.4 Threats to Validation

There are several threats to validity in our empirical studies.

First, our experiments are based on only 8 open-source software. More subject programs are desired to make the results more general.

Second, the faults should be extended to more real faults, not only with those injected ones.

At last, we should increase the types of faults. For example, it is appealing to study the multiple faults in one program, or faults that are related to more than one statements.

6. RELATED WORKS

There are several works that are related to our study, and they can be categorised into two types:

The first type of methods studies the MFS in CT. Nie [22] firstly studied the properties of the MFS in SUT, based on which additional test cases were generated to identify them. Other approaches to identify the MFS in SUT include building a tree model [32], adaptively generating additional test cases according to the outcome of the last test case [35], ranking suspicious interactions based on some rules [10], and using graphic-based deduction [19], among others. These approaches can be partitioned into two categories [5] according to how the additional test cases are generated: *adaptive*–additional test cases are chosen based on the outcomes of the executed tests [28, 22, 10, 24, 35, 27, 31, 17]or *nonadaptive*–additional test cases are chosen independently and can be executed in parallel [32, 5, 19, 20, 34]. All these works focused on input-level or configuration-level testing, i.e., their target is to identify the failure-inducing inputs or options, not the code in the program.

Besides these work, there exists two studies that focus on utilizing MFS to locate the faulty statement in the program [18, 8]. The first work [18] adopted OFOT [22] to calculate the MFS, and then compares the additional generated test cases with the original failing test cases to locate the failure-causing chains and corresponding statements. The second work [8] took the ranking method [10] to identify the MFS. Then it generated additional passing test cases based on the MFS, and ranked the suspicious statements by comparing the spectrum of the failing test case and passing test cases.

Our work differs from the first type of work in that we focus on both MFS and faulty code, not just any single object. Additionally, our work does not specifically describes how to identify MFS or how to conduct code-level fault localization, instead, we studied the correlation between them. We believe this is important and will give a guideline for how to utilize MFS for fault diagnosis.

The second type of work relates to the guaranteed code.

Elnatan [25] firstly used symbolic evaluation to analyse the configuration options in the program. In that work, they proposes the notion of guaranteed coverage, from which they find that the effective configuration space is relatively small when compared to the all the possible combinations of options. What's more, most coverage was accounted for by lower-strength interactions (2-way or 3-way), across all of line, basic block, edge, and conditions, but higher-strength interactions are needed for maximum coverage. Based on this work, Charles [29] proposed a test cases generation method, called iTree, which combines covering array and machine learning techniques to discover as more new coverage interactions as possible. Their experiments shows that their method is more effective than traditional CT and random methods at generating test cases with more coverage of program statements. They further refined their method iTree [30] by re-constructing the composite interactions.

Our work differs from them in 1) we only focus on the guaranteed code of the MFS instead of all the possible interactions in the SUT. 2) we mainly studied the guaranteed code for fault localization instead of test case generation and program statements coverage.

7. CONCLUSIONS AND FUTURE WORKS

Combinatorial testing has been proven to be effective at detecting and identifying the failure-inducing interactions, i.e., MFS in the SUT. Most of their work focus on how to generate test cases and how to effectively identify the MFS. Few of them consider the relationship between MFS and code-level problem. In this paper, we studied the correlation between MFS and faulty code. Specifically, we obtain the weak and strong guaranteed code of MFS, and then compare them with faulty code to observe their relationship. Our empirical studies suggest that it do exists correlation between MFS and faulty code, but the extent to which of this relation depends on the faulty types and inputs modeling.

As a further work, we plan to use the conclusion in this paper to utilize MFS for code-level fault diagnosis. It is also appealing to study whether code-level fault diagnosis can optimal the inputs modeling of CT, according to the our 3rd empirical study. Besides them, we would like to conduct studies on more software programs with more types of faults to increase the generality of our work.

8. ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (No. 61272079), the Research Fund for the Doctoral Program of Higher Education of China (No.20130091110032), the Science Fund for Creative Research Groups of the National Natural Science Foundation of China(No. 61321491), the Major Program of National Natural Science Foundation of China (No. 91318301), and National Science Foundation Award CCF-1464425.

9. REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE, 2007.

[2] R. Chillarege. Orthogonal defect classification. *Handbook of Software Reliability Engineering*, pages 359–399, 1996.

[3] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The aetg system: An approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23(7):437–444, 1997.

[4] M. B. Cohen, C. J. Colbourn, and A. C. Ling. Augmenting simulated annealing to build interaction test suites. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 394–405. IEEE, 2003.

[5] C. J. Colbourn and D. W. McClary. Locating and detecting arrays for interaction faults. *Journal of combinatorial optimization*, 15(1):17–48, 2008.

[6] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[7] J. A. Duraes and H. S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.

[8] L. S. Ghandehari, Y. Lei, D. Kung, R. Kacker, and R. Kuhn. Fault localization based on failure-inducing combinations. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 168–177. IEEE, 2013.

[9] L. S. G. Ghandehari, M. N. Bourazjany, Y. Lei, R. N. Kacker, and D. R. Kuhn. Applying combinatorial testing to the siemens suite. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 362–371. IEEE, 2013.

[10] L. S. G. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker. Identifying failure-inducing combinations in a combinatorial test set. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 370–379. IEEE, 2012.

[11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 191–200. IEEE, 1994.

[12] Y. Jia, M. B. Cohen, M. Harman, and J. Petke. Learning combinatorial interaction test generation strategies using hyperheuristic search. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 540–550. IEEE Press, 2015.

[13] 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*, pages 273–282. ACM, 2005.

[14] D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, pages 91–95. IEEE, 2002.

[15] D. R. Kuhn, D. R. Wallace, and J. AM Gallo. Software fault interactions and implications for software testing. *Software Engineering, IEEE Transactions on*, 30(6):418–421, 2004.

[16] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence. Ipog/ipog-d: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability*, 18(3):125–148, 2008.

[17] J. Li, C. Nie, and Y. Lei. Improved delta debugging based on combinatorial testing. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 102–105. IEEE, 2012.

[18] C. Ma, Y. Zhang, J. Liu, et al. Locating faulty code using failure-causing input combinations in combinatorial testing. In *Software Engineering (WCSE), 2013 Fourth World Congress on*, pages 91–98. IEEE, 2013.

[19] C. Martínez, L. Moura, D. Panario, and B. Stevens. Algorithms to locate errors using covering arrays. In *LATIN 2008: Theoretical Informatics*, pages 504–519. Springer, 2008.

[20] C. Martínez, L. Moura, D. Panario, and B. Stevens. Locating errors using elas, covering arrays, and adaptive testing algorithms. *SIAM Journal on Discrete Mathematics*, 23(4):1776–1799, 2009.

[21] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11, 2011.

[22] C. Nie and H. Leung. The minimal failure-causing schema of combinatorial testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):15, 2011.

[23] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11, 2011.

[24] X. Niu, C. Nie, Y. Lei, and A. T. Chan. Identifying failure-inducing combinations using tuple relationship. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 271–280. IEEE, 2013.

[25] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 445–454. ACM, 2010.

[26] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 30–39. IEEE, 2003.

[27] K. Shakya, T. Xie, N. Li, Y. Lei, R. Kacker, and R. Kuhn. Isolating failure-inducing combinations in combinatorial testing using test augmentation and classification. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 620–623. IEEE, 2012.

[28] L. Shi, C. Nie, and B. Xu. A software debugging method based on pairwise testing. In *Computational Science–ICCS 2005*, pages 1088–1091. Springer, 2005.

[29] C. Song, A. Porter, and J. S. Foster. itree: Efficiently discovering high-coverage configurations using interaction trees. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 903–913. IEEE Press, 2012.

[30] C. Song, A. Porter, and J. S. Foster. itree: efficiently discovering high-coverage configurations using interaction trees. *IEEE Transactions on Software Engineering*, 40(3):251–265, 2014.

[31] Z. Wang, B. Xu, L. Chen, and L. Xu. Adaptive interaction fault location based on combinatorial testing. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 495–502. IEEE, 2010.

[32] C. Yilmaz, M. B. Cohen, and A. A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on*, 32(1):20–34, 2006.

[33] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.

[34] J. Zhang, F. Ma, and Z. Zhang. Faulty interaction identification via constraint solving and optimization. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 186–199. Springer, 2012.

[35] Z. Zhang and J. Zhang. Characterizing failure-causing parameter interactions by adaptive testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 331–341. ACM, 2011.