Is MFS Strongly Correlated with faulty code? *

Xintao Niu
State Key Laboratory for Novel Software Technology
Nanjing University
China, 210023
niuxintao@gmail.com

Changhai Nie
State Key Laboratory for Novel Software Technology
Nanjing University
China, 210023
changhainie@nju.edu.cn

Xiaoyin Wang
Department of Computer Science
The University of Texas at San Antonio
Xiaoyin.Wang@utsa.edu

Hareton Leung
Department of computing
Hong Kong Polytechnic University
Kowloon, Hong Kong
hareton.leung@polyu.edu.hk

Jeff Lei
Department of Computer Science and Engineering
The University of Texas at Arlington
ylei@cse.uta.edu

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, Symbolic execution, Failure-inducing interactions, Guaranteed code

*(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

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 [6, 7]. 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[6]. 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 [1, 2, 8, 5], or to identify the MFS more accurately [11, 12, 17, 13]. 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 [14]. Obtaining the guaranteed code of an interaction can help us understand how this interaction influence on the be-

haviour 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 [12], 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 [14], which firstly 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 [14] 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

This section presents some formal definitions 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)* [12] 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 [3, 10, 11], 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, -).

```
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

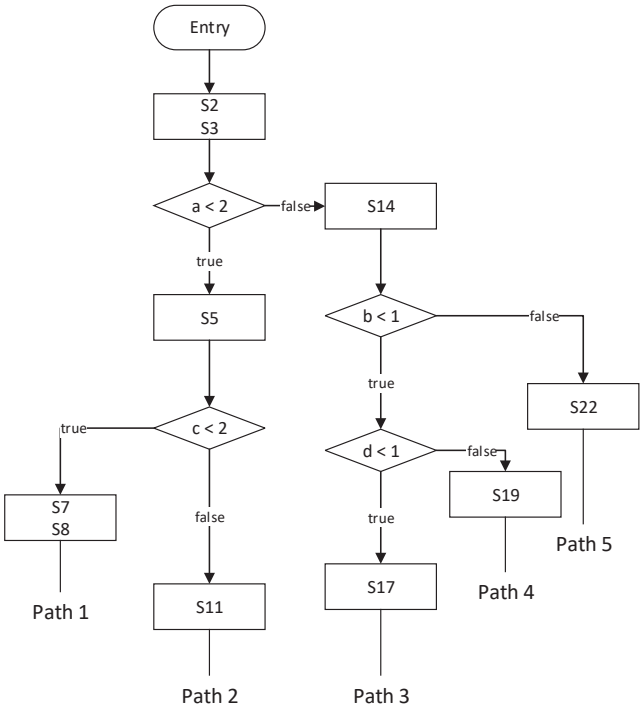

Figure 2: The execution tree of program *foo*

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

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 [12], which is a well-known MFS identification method in CT. In this paper, we will focus on this identification method.

2.3 Basic definitions about guaranteed code

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 [15]. Let $Pcon(\mathcal{P}) = < p_1, p_2, p_3, ...p_k >$ 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.

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	$\neg a < 2$, $\neg b < 1$

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

Definition 5. For a schema c, and a path $\mathcal{P}$, if all the path conditions in this path can be satisfied when this schema is contained in the input, i.e., $(\bigwedge_{p_i \in Pcon(\mathcal{P})} p_i) \bigwedge c = true$, we call $\mathcal{P}$ the *Consistent path* of schema c.

In fact, *consistent path* of one schema is a path can be reached when given one input which contains this schema. For example,

Based on this definition, we use the notation $CsP(c)$ to represent the set of all the consistent paths of schema c, i.e., $CsP(c) = \{\mathcal{P}_i | (\bigwedge_{p_i \in Pcon(\mathcal{P}_i)} p_i) \bigwedge c = true\}$. 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, $wg(c) = \bigcap_{\mathcal{P} \in CsP(c)} Cov(\mathcal{P})$.

Essentially, the weak guaranteed code[1] of one schema, 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.

As an example,

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 there may exists some other schemas that control some part of this weak guaranteed code, but these schemas may always appears with the corresponding schema. Obviously, these schemas are the sub-schemas of this schema. For example, in Fig, .

Hence, 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, $sg(c) = wg(c) \backslash \{\bigcup_{c' \prec c} wg(c')\}$.

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, .

In this paper, we focus on the guaranteed code of MFS, instead of all the other schemas in the test case. With respect to this example, it is easily find that the weak guaranteed code and strong guaranteed code is, respectively. They are very correlated to the real faulty code, especially for the strong guaranteed code, . However, in the real situation, it may . As we want to know whether to detect and identify the MFS is really helpful in the code-based diagnosis, we need to do more empirical studies to find the result.

3. RESEARCH QUESTIONS

To comprehensive study the correlativeness between the MFS and real

3.1 The correlation between MFS and faulty code

Hence, it motivates our first research question:

Q1: Is MFS correlated to the faulty code?

This question is key to. We let the guaranteed code of MFS.

3.2 The influence of different types of faults

Q2: Different fault type?

Taks as an example.

3.3 The influence of inputs model

Q3: Is the building model affected MFS ?

Taks as an example.

These questions are important, as

4. SUBJECT PROGRAMS

4.1 Generating faulty programs

[1]The definition of weak guaranteed code is similar to the *guarantee coverage* defined in [14]. The difference is that in this paper we do not distinguish the input and value symbolic; instead, they are handled in an unified way.

4.2 Symbolic execution of the paths

5. RESULTS

5.1 Threats to Validation

6. RELATED WORKS

There are many methods aims at identifying MFS in CT, which can be classified into two categories:

1. MFS in CT

All these methods. with respect to the code level which utilizes. 2. use MFS to isolate faulty code. machunhyan [9] and leiyu [4]

Studies on symbolic execution, has .

4. Use symoblic execution in CT. Charles song [14] first. Then extend to itree [15], then extends to multiple leaves with [16].

Our work differs from them

7. CONCLUSIONS AND FUTURE WORKS

8. REFERENCES

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

APPENDIX