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 [5, 6]. 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[5]. 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 [2, 3, 7, 4], or to identify the MFS more accurately [8, 9, 14, 10]. 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 [12]. 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 [9], 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 [12], 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 [12] 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 definitions and propositions to give a formal model for CT.

2.1 Failure-inducing interactions in 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). The definitions below are originally defined in [9].

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.

We consider any abnormally executing test case as a *fault*. It can be a thrown exception, compilation error, assertion failure or constraint violation. When faults are triggered by some test cases, it is desired to figure out the cause of these faults.

Definition 2. For the SUT, the n-tuple (-,v_{n_1},...,v_{n_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 "-".

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)* [9] 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. Figuring the MFS out helps to identify the root cause of a failure and thus facilitate the debugging process.

2.2 CT Test Case Generation

When applying CT, the most important work is to determine whether the SUT suffers from the interaction faults or not, i.e., to detect the existence of the MFS. Rather than impractically executing exhaustive test cases, CT commonly design a relatively small set of test cases to cover all the schemas with the degree no more than a prior fixed number, t. Such a set of test cases is called the *covering array*. If some test cases in the covering array failed in execution, then the interaction faults is regard to be detected. Let us formally define the covering array.

Definition 5. $MCA(N; t, n, (a_1, a_2, ..., a_n))$ is a t-way *covering array* in the form of $N \times n$ table, where each row represents a *test case* and each column represents a parameter. For any t columns, each possible t-degree interaction of the t parameters (schema) must appear at least once. When $a_1 = a_2 = ... = a_n = v$, a t-way covering array can be denoted as $CA(N; t, n, v)$.

Table 1: A covering array

ID	Test case			
t_1	0	0	0	0
t_2	0	1	1	1
t_3	1	0	1	1
t_4	1	1	0	1
t_5	1	1	1	0

For example, Table 1 shows a 2-way covering array CA(5; 2, 4, 2) for the SUT with 4 boolean parameters. For any two columns, any 2-degree schema is covered. Covering array has proven to be effective in detecting the failures caused by interactions of parameters of the SUT. Many existing algorithms focus on constructing covering arrays such that the number of test cases, i.e., N, can be as small as possible. In general, most of these studies can be classified into three categories according to the construction strategy of the covering array:

1) One test case one time : This strategy repeats generating one test case as one row of the covering array and counting the covered schemas achieved until all schemas are covered [2, 1, 13].

2) A set of test cases one time: This strategy generates a set of test cases at each iteration. Through mutating the values of some parameters of some test cases in this test set, it focuses on optimising the coverage. If the coverage is finally satisfied, it will reduce the size of the set to see if fewer test cases can still fulfil the coverage. Otherwise, it will increase the size of test set to cover all the schemas[3, 11].

3) IPO-like style: This strategy differentiates from the previous two strategies in that it does not firstly generate complete test cases [7]. Instead, it first focuses on assigning values to some part of the factors or parameters to cover the schemas that related to these factors, and then fills up the remaining part to form complete test cases.

In this paper, we focus on the first strategy: One test case one time as it immediately get a complete test case such that the testers can execute and diagnose in the early stage. As we will see later, with respect to the MFS identification, this strategy is the most flexible and efficient one comparing with the other two strategies.

2.3 MFS

Definition
Getting approaches
The details is as follows:

2.4 Symbolic execution

2.5 guaranteed code code

definition guaranteed code
definition minimal guaranteed code

For example, from Figure 2(b) we can see that any configuration satisfying (i.e., a=0, b=1) is guaranteed to cover L2, no matter the choice of and N. We can use Otter's output to compute the guaranteed coverage for a predicate p, which we will write as Cov(p). We first Lnd CovT (p), the coverage guaranteed under p by test case T, for each test case; then, Cov(p) = ST CovT (p). To compute CovT (p), let pTi be the path conditions from T's symbolic evaluation, and let CT (pTi) be the covered lines (or blocks, edges, conditions, etc.) that occur in that path. Then CovT (p) is ConsistentT (p) = fpTi j SAT(pTip)g CovT (p) = T q2ConsistentT (p) CT (q). In other words, first we compute the set of path conditions pTi such that p and pTi are consistent. If this holds for pTi, the items in CT (pTi) may be covered if p is true. Since our symbolic evaluator explores all possible program paths, the intersection of these sets for all such pTi is the set guaranteed to be covered if p is true.

Going back to Figure 2, here are some predicates and the coverage they guarantee given the test cases input=1 and input=0. We abbreviate K = 1 as K and K = 0 as :K.

In this paper, we use the symbolic execution to . The details is as follwos:

3. RESEARCH QUESTION

3.1 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

4.2 symbolic execution of the paths

5. RESULTS

5.1 Threats to Validation

6. RELATED WORKS

7. CONCLUSIONS AND FUTURE WORKS

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] R. C. Bryce and C. J. Colbourn. The density algorithm for pairwise interaction testing. *Software Testing, Verification and Reliability*, 17(3):159–182, 2007.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] K. J. Nurmela. Upper bounds for covering arrays by tabu search. *Discrete applied mathematics*, 138(1):143–152, 2004.

[12] 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.

[13] Y.-W. Tung and W. S. Aldiwan. Automating test case generation for the new generation mission software system. In *Aerospace Conference Proceedings, 2000 IEEE*, volume 1, pages 431–437. IEEE, 2000.

[14] 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