Error Locating Driven Array*

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

JiaXi Xu
School of Mathematics and Information Technology
Nanjing Xiaozhuang University
China, 211171
xujiaxi@126.com

ABSTRACT

Combinatorial testing(CT) seeks to handle potential faults caused by various interactions of factors that can influence the Software systems. When applying CT, it is a common practice to first generate a bunch of test cases to cover each possible interaction and then to locate the failure-inducing interaction if any failure is detected. Although this conventional procedure is simple and straightforward, we conjecture that it is not the ideal choice in practice. This is because 1) testers desires to isolate the root cause of failures before all the needed test cases are generated and executed 2) the early located failure-inducing interactions can guide the remaining test cases generation, such that many unnecessary and invalidate test cases can be avoided. For this, we propose a novel CT framework that allows for both generation and localization process to better share each other's information, as a result, both this two testing stages will be more effectively and efficiently when handling the testing tasks. We conducted a series of empirical studies on several open-source software, of which the result shows that our framework can locate the failure-inducing interactions more quickly than traditional approaches, while just needing less test cases.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and debugging—*Debugging aids,testing tools*

General Terms

Reliability, Verification

*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), and the Major Program of National Natural Science Foundation of China (No. 91318301)

WOODSTOCK '97 El Paso, Texas USA

Keywords

Software Testing, Combinatorial Testing, Covering Array, Failure-inducing combinations

1. INTRODUCTION

Modern software is developed more sophisticatedly and intelligently than before. 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 compatibility problems. In consideration of the scale of the real industrial software, to test all the possible combination of all the factors (we call them the interaction space) is not feasible, and even it is possible, it is not recommended to test exhaustive interactions for most of them do not provide any useful information.

Many empirical studies shows that in real software systems, the effective interaction space, i.e., targeting fault defects, makes up only a small proportion of the overall interaction space. What's more, the number of factors involved in these effective interactions is relatively small, of which 4-6 is usually the upper bonds. With this observation, applying CT in practice is appealing, as it is proven to be effective to handle the interaction faults in the system.

A typical CT life-circle is listed as Figure 1, in which there are four main testing stages. At the very beginning of the testing, engineers should extract the specific model of the software under test. In detail, they should identify the factors, such as user inputs, configure options, environment states and the like that could affect the system's behavior. Next, which values each factor can take should also be determined. Further efforts will be needed to figure out the constraints and dependencies between each factor and corresponding values to make the testing work valid. After the modeling stage, a bunch of test cases should be generated and executed to expose the potential faults in the system. In CT, one test case is a set of assignment of all the factors that we modeled before. Thus, when such a test case is executed, all the interactions contained in the test case are deemed to be checked. The main target of this stage is to design a relatively small size of test cases to get some specific coverage, for CT the most common coverage is to check all the possible interactions with the number of factors no more than a prior fixed integer, i.e., strength t. The third testing stage in this circle is the fault localization, which is responsible for diagnosing the root cause of the failure we detected before. To characterize such root cause, i.e., failure-inducing

interactions of corresponding factors and values is important for future bug fixing, as it will reduce the suspicious code scope that needed to inspect. The last testing stage of CT is evaluation. In this stage, testers will assess the quality of the previously conducted testing tasks, many metrics such as whether the failure-inducing interactions can reflect the failures detected, whether the generated test cases is adequate to expose all the behaviors of the system, will be validated. And if the assessment result shows that the previous testing process does not fulfil the testing requirement, some testing stages should be made some improvement, and sometimes, may even need to be re-conducted.

Although this conventional CT framework is simple and straightforward, however in terms of the test cases generation and fault localization stages, we conjecture that the first-generation-then-localization is not the proper choice for most test engineers. This is because, first, it is not realistic for testers wait for all the needed test cases are generated before they can diagnosis and fix the failures that haven been detected, second, and the most important, utilizing the early determined failure-inducing interactions can guide the following test cases generations, such that many unnecessary and invalid test cases will be avoided. For this we get to the most key idea of this paper:

Generation and Localization process should be organised in a more tightly way.

Based on the idea, we propose a new CT framework, which instead of dividing the generation and localization into two independent stages, it integrate this two stages into one. We call this new one the Generation-Localization stage, which allows for both generation and localization better share each other's testing information. To this aim, we remodel the generation and localization modular to make them better adapt to this newly framework. In specific, our generation adopts the one-test-one-time strategy, i.e., generate and execute one test case in one iteration. Rather than generating the overall needed test cases in one time, this strategy is more flexible so that it allows for terminating at any time during generation, no matter whether the coverage is reached or not. With this strategy, we can let the generation stops at the point we detect some failures, and then after localization we can utilize the diagnosing result to change the coverage criteria, e.g., the interactions related to the failure-inducing interactions do not need to be checked any more. Then based on the new coverage criteria, the generation process goes on.

We conducted a series empirical studies on several open-source software to evaluate our newly framework. In short, we take two main comparisons, one is to compare our new framework with the traditional one, which first generate a complete set of test cases and then perform the fault localization. Another one is to compare with the Error locating array, which is a well-designed set of test cases that can be used directly detect and isolate the failure-inducing interactions. The results shows that in terms of test case generation and fault diagnosis, our approach can and significantly reduced the overall needed test cases and can more quickly isolate the root cause of the system under test.

The main contributions of this paper:

1. We propose a newly CT framework which combine the test cases generation and fault localization more closely.

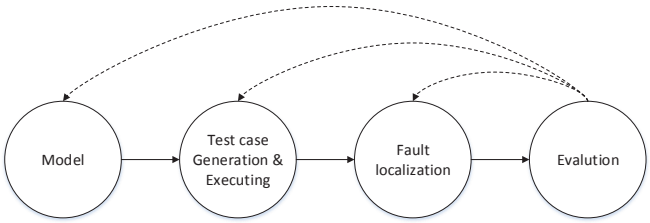

Figure 1: The life circle of CT

2. We augment the traditional CT test cases generation and fault localization process to make them adapt to the newly framework.

3. A series comparisons with traditional CT is conducted and the result of the empirical studies are discussed.

The reset of the paper is organised as follows: Section 2 presents the preliminary background of some definitions of the CT. Section 3 describes our newly framework and a simple case study is also given. Section 4 presents the empirical studies and the results are discussed. Section 5 shows the related works. Section 6 concludes the paper and propose some further works.

2. BACKGROUND

This section presents some definitions and propositions to give a formal model for the FCI problem.

Assume that the SUT is influenced by n parameters, and each parameter p_i has a_i discrete values from the finite set V_i, i.e., $a_i = |V_i|$ (i = 1,2,..n). Some of the definitions below are originally defined in .

Definition 1. A *test case* of the SUT is an array of n values, one for each parameter of the SUT, which is denoted as a n-tuple (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 software.

We consider the fact that the abnormally executing test cases as a *fault*. It can be a thrown exception, compilation error, assertion failure or constraint violation. When faults are triggered by some test cases, what is desired is to figure out the cause of these faults, and hence some subsets of this test case should be analysed.

Definition 2. For the SUT, the n-tuple (-,v_{n_1},...,v_{n_k},...)is called a k-value *combination* ($0 < k \leq n$) when some k parameters have fixed values and the others can take on their respective allowable values, represented as "-".

In effect a test case itself is a k-value *combination*, when k = n. Furthermore, if a test case contain a *combination*, i.e., every fixed value in the combination is in this test case, we say this test case *hits* the *combination*.

Definition 3. let c_l be a l-value combination, c_m be an m-value combination 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-combination* of c_m and c_m is a *parent-combination* of c_l, which can be denoted as $c_l \prec c_m$.

For example, in the motivation example section, the 2-value combination (-, 4, 4, -) is a sub-combination of the 3-value combination (-, 4, 4, 5), that is, (-,4,4,-) $\prec$ (-,4,4,5).

Definition 4. If all test cases contain a combination, say c, trigger a particular fault, say F, then we call this combination c the *faulty combination* for F. Additionally, if none sub-combination of c is the *faulty combination* for F, we then call the combination c the *minimal faulty combination* for F (It is also called Minimal failure-causing schema(MFS) in).

In fact, MFS and *minimal faulty combinations* are identical to the failure-inducing combinations we discussed previously. Figuring it out can eliminate all details that are irrelevant for causing the failure and hence facilitate the debugging efforts.

2.1 Detect the failure-inducing schemas

When applying CT on software testing, the most important work is to determine whether the SUT is suffering 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 size 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 calling the covering array. If some test cases in the covering array failed after execution, then the interaction faults is regard to be detected.

Many studies in CT field focus on how to generate such a test suite with the aim that making the size of the test suite as small as possible. In general, most of these studies can be classified into three categories according to the construction way 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 so far until no more schemas is needed to be covered.

2) A overall set of test cases one time: This strategy first generates a set of test cases with the size of the set fixed in prior. Then through some operation such as mutation of the some cells, increase or decrease the size of the set of test cases, regeneration some test cases to make the set of test cases to cover all the needed schemas with the size as small as possible.

3) Others : This strategy differentiates from the previous two strategies at the point it does not first give completed test cases. It will first focus on assigning values to some particular factors or parameters to cover the schemas that related to these factors, and then complement the remains to form completed test cases.

In this paper, we focus on the first one: One test case one time as it can allow for immediately getting a completed test case such that the testers can execute and diagnosis in the early stage. And we will see later, with respect to the fault defeating, this strategy us the most flexible and efficient one comparing with the other two strategies.

2.2 Isolate the failure-inducing schemas

To detect the existence of MFS in the SUT is still far from figuring out the root cause of the failure. As we do not know exactly which one or some schemas in the failed test cases should be responsible for the failure. In fact, for a failing test case, there can be at most $2^k - 1$ possible schemas can be the MFS. Hence, further fault diagnosis is desired, i.e., more test cases should be generated to isolate the MFS.

A typical MFS isolating process is as Table 1. This example assumes the SUT has 3 parameters, each can take 2 values. And assume the test case (1, 1, 1) failed. Then in Table 1, as test case t failed, and OFOT mutated one factor of the t one time to generate new test cases: $t_1 - t_3$, It found the the t_1 passed, which indicates that this test case break the MFS in the original test case t. So the (1,-,-) should be one failure-causing factor, and as other mutating process all failed, which means no other failure-inducing factors were broken, therefore, the MFS in t is (1,-,-).

Table 1: OFOT with our strategy

original test case				**Outcome**
t	1	1	1	Fail
observed				
t_1	0	1	1	Pass
t_2	1	0	1	Fail
t_3	1	1	0	Fail

This isolation process mutate one factor of the original test case one time to generate extra test cases. And then according to the outcome of the test cases execution result, it will identify the MFS of the original failing test cases. It is calling the OFOT method, which is the well-known fault diagnosis method in CT. In this paper, we will focus on this isolation method. It is noted that our following proposed new CT framework can be easily applied on other CT fault diagnosis methods.

3. THE INTEGRATED GENERATION FAULT LOCALIZATION PROCESS

As we discussed previously, the generation and localization is the most key work in CT life-circle. How to utilize this two work in the CT life-circle is of importance as it is closely related to the quality and cost of overall software testing. In fact, most works in CT focus on this two fields, however, they are almost discussed independently, i.e., either only aims to improve the generation efficiency or only devote to refining the localization process.

focus interesting and many existed works should. However, most of these works is independently to describe, better improve. Less of them how to and when to use them together, they think it is of no sense, as the first-generation-then-isolation is so natural and straightforward. However, as we found the following, how to utilize them really make sense.

3.1 traditional generation-isolation process

To isolate the failure-inducing schemas in SUT, a typical traditional generation-isolation life-circle is to generate a t-way covering array to detect if there exists some failures that triggered by some particular schemas. And after the generation and execution, if we detect some failures, we should isolate the failure-inducing schemas in the SUT. And further diagnosis operation should be taken to bug fixing.

As listed in Table, which.

Such life-circle is not the proper choice. The first reason we had discussed previous, which is the engineers should be too patient to wait for bug fixing. And also as we discussed previously, the second reason should be the most important, which we use an example to illustrate the reason. A typical example is like the following, for which we can find some duplicated. It is noted that some duplicated test cases can be avoided by . Thus, a more effective and efficient framework is desired.

3.2 the new framework

Our new framework is as illustrated.

3.3 Description

3.4 A case study

4. EMPIRICAL STUDIES

4.1 The existence of

4.1.1 Study setup

4.1.2 Result and discussion

4.2 Performance of the traditional algorithms

4.2.1 Study setup

4.2.2 Result and discussion

4.3 Comparing with Error locating Array

Erray locating array is a well-desinged test cases.

4.3.1 Study setup

4.3.2 Result and discussion

4.4 Threats to validity

5. RELATED WORKS

6. CONCLUSIONS