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.

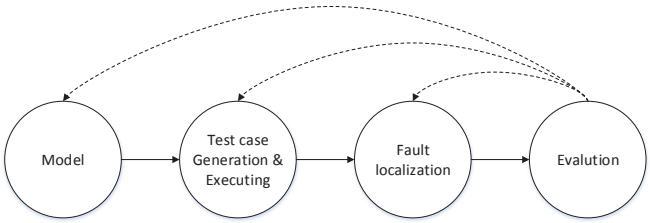

Figure 1: The life circle of CT

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 works 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, in which they are almost discussed independently, i.e., either only aims to improve the generation efficiency or only devotes to refining the localization process. The justification for not discussing how to cooperate this two works is that they think the first-generation-then-isolation is so natural and straightforward. As we will show below, however, that the generation and localization is so tightly correlated and how to cooperate this two works do have an significant impact on the effectiveness and efficiencies of the testing work.

3.1 traditional generation-isolation process

A typical traditional generation-isolation life-circle is to first generate a t-way covering array to detect if there exists some failures that triggered by some particular schemas. Then if we detect some failures, we should isolate the failure-inducing schemas in the SUT for further bug fixing.

As an example, Table 2, which illustrate the process of testing the System with 3 parameters and each parameter

has two values. It first generated and executed the 2-way covering array (t_1 – t_4). And after finding that t_1 and t_2 fails during testing, it then respectively isolate the MFS in the t_1 and t_2. For the t_1, it uses OFOT method generates three additional test cases (t_5 – t_7), and identified the MFS of t_1 is (0, - , -) as only when changing the first factor of t_1 it will pass. Then it will do the same thing to t_2, and found that (0, -, -) is also the MFS of t_2. It must be noted that, when isolating the MFS of t_2, the generated extra test cases t_9 and t_{10} has appeared before, so we can just reuse the previous result. After all, for detecting and isolate the MFS in this example SUT, we have use 8 test cases(marked with star).

Table 2: traditional generation-isolation life-circle

generation and execution				
	test case			**Outcome**
t_1*	0	0	0	Fail
t_2*	0	1	1	Fail
t_3*	1	1	0	Pass
t_4*	1	0	1	Pass
localization				
for $t1$—	**0**	**0**	**0**	
t_5*	1	0	0	Pass
t_6*	0	1	0	Fail
t_7*	0	0	1	Fail
	result – (0, -, -)			
for $t2$ —	**0**	**1**	**1**	
t_8*	1	1	1	Pass
t_9	0	0	1	Fail
t_{10}	0	1	0	Fail
	result – (0, -, -)			

Such life-circle is not the proper choice in practice. The first reason we had discussed previously is that the engineers normally cannot be so patient to wait for fault localization when failure is found. The early bug fixing is appealing and can give the engineers confidence to keep on improving the quality of the software. The second reason, which is also the most important, is such life-circle can generate many redundant and unnecessary test cases. This can be reflected in the following two aspects:

1) The test cases generated in the localization stage can also contribute some coverage, i.e., the schemas appear in the passing test cases in the localization stage may have already been covered in the test cases generation and execution stage. For example, when we identify the MFS of t_1 in Table 2, the schema (-, 1, 0) contained in the extra passing test case t_6 – (0 1 0) has already been appeared in the passing test case t_3 – (1 1 0). In another word, if we firstly isolate the MFS of t_1, then the t_3 is not the good choice as it doesn't covered as many as possible 2-value schemas, say, (1 1 1) is better than this test case at contributing more coverage.

2)The identified MFS should not appeared in the following generated test cases. This is because for the definition of MFS, each test case contain this schema will trigger a failure, i.e., to generate and execute more than one test case contained the MFS makes no sense for the failure detecting. Worse more, such test case will cause some masking effects, as it such test case may make other schemas in the test case omitted in the testing for the failure it trigged. For example, when we identify the MFS of t_1 and find that (0 - -) is the failure-inducing schema, we should not generate the test case t_2, because it also contain this schema, it must also fail during testing as expected. And since we should not generate t_2, then the test cases generated for identified the MFS in t_2 is also needless.

For all of this, a more effective and efficient framework is desired.

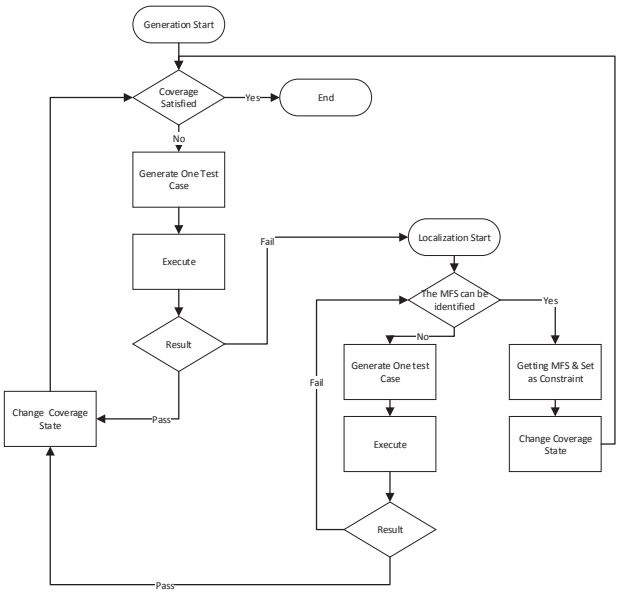

Figure 2: New Framework of CT

3.2 the new framework

To handle such deficiencies in traditional CT, we propose a new CT generation-localization framework. Our new framework aims at strengthening the interaction of generation and localization to reduce the unnecessary and invalid test cases discussed previously. The basic outline of our framework is illustrated in Figure 2.

In specific, this new framework works as follows: First, it will check whether all the needed schemas is covered or not. Commonly the target of CT is to cover all the t-valued schemas, with t normally be assigned to be 2 or 3. Then if the coverage currently is not satisfied, it will generate a new test case to cover the schemas that is still not be covered as more as possible. After that, it will execute this test case with the outcome of the execution either be pass (executed normally, i.e., doesn't triggered an exception, violate the expected oracle or the like) or fail (on the contrary). When the test case pass the execution, we will recompute the coverage state, as all the schemas in the passing test case are regarded as the error-irrelevant. As a result, the schemas that wasn't covered before will be determined to be covered if it is contained in this newly generated test case. Otherwise if the test case fails, then we will start the MFS identify modula, to isolate the MFS in this failing test case. One point that needs to be noted is that if the test case fails, we will not change the coverage state, as we can not figure out which schemas are responsible to this failure among all the schemas in this test case until we isolate them.

As for the identify modula, it keeps almost the same as traditional independent MFS identify process, i.e., repeats generating and executing additional test cases until it can get enough information to diagnosis the MFS in the original failing test case. The only difference from traditional MFS identifying process is that we augment it by counting the coverage this modula have to the overall coverage. In detail, when the additional test case passes, we will label the schemas in these test cases as covered if it is not covered yet. And when the MFS is found at the end of this modula, we will first set them as forbidden schemas that latter generated test cases should not contained,(Otherwise the test case must fail and it cannot contribute to more coverage), second all the schemas that contain this MFS will be set as covered. (As the MFS will not appear again, these schemas contain this MFS will also not be generated, so we will make them as covered to let the generation process stoppable)

Next we will specifically describe the two important parts in this framework:

1) *Generation* : We adopt the one-test-case-one-time method as the basic skeleton of the generation process. And as shown in Figure 1, we should account for the MFS to let them not appear in the latter generated test cases, which should be handled as the constraints–forbidden tuples. Our test case generation with consideration for constraints is inspired by the Cohen's AETG-SAT, based on which we give an more general approach that can be applied on more one-test-one-time generation methods. The detail of this modula is described in Algorithm 1.

Algorithm 1 Generate One test Case

Input: $Param$ ▷ values set that each option can take
S_{MFS} ▷ the set of MFS that currently isolated
$t_{uncovered}$ ▷ the schemas that are still uncovered
Output: $test$ ▷ the generate test case

1: $Candidate_{test} \leftarrow emptySet$
2: **while** $Candidate_{test}.size$ is not satisfied **do**
3: $test \leftarrow emptyArray(Param.size)$
4: **for each** $factor \in test$ **do**
5: $value \leftarrow select(t_{uncovered}, Param, opt)$
6: **while not** $CheckSat(test, S_{MFS})$ **do**
7: $value \leftarrow repick(t_{uncovered}, Param, opt)$
8: **end while**
9: $test.set(factor, value)$
10: **end for**
11: $Candidate_{test}.append(test)$
12: **end while**
13: $best \leftarrow select(Candidate_{test})$
14: **return** $best$

This algorithm first gives a candidate set which initially is set to be empty(line 1). We lastly will fill up the set and select a best test case according to some criteria(line 13). For greedy algorithms like AETG this criteria may be the test case which contain the most uncovered schemas. For each test case that will be appended in this candidate set, it must satisfied two conditions: 1. The value each factor be assigned should optimize some coverage criteria (line 5), 2. The values currently be assigned to corresponding factors should **not** lead the test case eventually contain the some MFS (line 6 - 8).

It is noted that we didn't specify that in which order these factors should be assigned and the value selection criteria for each factor is also not shown. This is because for different One-test-one-time generation methods, these metrics varies from each other. Another point that needs to be noted is that the constraints checking part(line 6 - 8) of our algorithm does not aim to optimising for the performance like running-time, iteration number or the like. Some studies, like Cohen in, by exploiting the SAT history or setting the threshold, can significantly improve such performance. In this paper, however, we will not discuss the details for those techniques. Instead, we want to make the overall generation process more general and fit for the framework listed in Figure 2.

2) *Isolation* : The isolation process will also be appropriately modified to adapt for the new CT life-circle. is still the same as before: 1) the additional test case should not contain the MFS, if cannot find, set the test as . 2) when update the MFS set. and coverage set.

Algorithm 2 replace test cases triggering unexpected faults

Input: $t_{original}$ ▷ original failing test case
S_{MFS} ▷ previously identified MFS
$t_{uncovered}$ ▷ the schemas that are still uncovered
$Param$ ▷ values set that each option can take

1: **while not** FindMFS() **do**
2: $s_{mutant} \leftarrow t_{original} - s_{fixed}$
3: **for each** $opt \in s_{mutant}$ **do**
4: $i = getIndex(Param, opt)$
5: $opt \leftarrow opt'$ $s.t.$ $opt' \in Param[i]$ and $opt' != opt$
6: **end for**
7: $t_{new} \leftarrow s_{fixed} \bigcup s_{mutant}$
8: $result \leftarrow execute(t_{new})$
9: **if** $result == PASS$ or $result == F_m$ **then**
10: **return** t_{new}
11: **else**
12: continue
13: **end if**
14: **end while**
15: $S_{MFS}.append(newMFS)$.
16: $t_{newCovered} \leftarrow FindingTschemas(S_{MFS})$
17: $t_{uncovered}.reduce(t_{newCovered})$

With this newly framework, when we re-consider the example in Table 2 we can get the following result listed in Table 3.

Table 3: newly generation-isolation life-circle generation and execution

	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

3.3 Description

3.4 A case study

4. EMPIRICAL STUDIES