Error Locating Driven Array*

Xintao Niu and Changhai Nie
State Key Laboratory for Novel Software Technology
Nanjing University, China
niuxintao@gmail.com
changhainie@nju.edu.cn

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

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)

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 to 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 framewbach2004pairwiseork 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 [9]; 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 module 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.

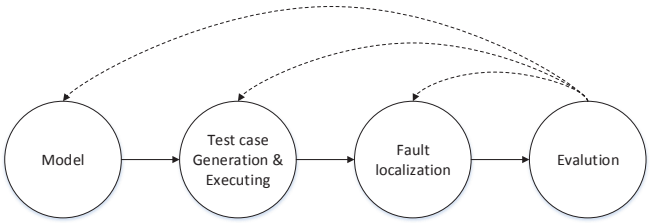

Figure 1: The life circle of CT

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 [2].

3) Others : This strategy differentiates from the previous two strategies at the point it does not first give completed test cases [6]. 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 remaining part 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 is 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 studies in CT focus on this two fields. But rather than as a whole, generation and localization are discussed independently. 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 4 parameters and each parameter has two values. It first generated and executed the 2-way covering array ($t_1 - t_9$). And after finding that t_1, t_2, and t_7

failed during testing, it then respectively isolated the MFS in the t_1 ,t_2, and t_7. For the t_1, it uses OFOT method generates four additional test cases ($t_{10} - t_{13}$), and identified the MFS of t_1 is (-, 0, - , -) as only when changing the second factor of t_1 it will pass. Then it will do the same thing to t_2 and t_7, and found that (-,0, -, -) is also the MFS of t_2 and t_7. After all, for detecting and isolate the MFS in this example SUT, we have generated 12 additional test cases(marked with star).

Table 2: traditional generation-isolation life-circle

generation					
	test case				**Outcome**
t_1	0	0	0	0	Fail
t_2	0	1	1	1	Pass
t_3	0	2	2	2	Pass
t_4	1	0	1	2	Fail
t_5	1	1	2	0	Pass
t_6	1	2	0	1	Pass
t_7	2	0	2	1	Fail
t_8	2	1	0	2	Pass
t_9	2	2	1	0	Pass
localization					
for $t1$— **0 0 0 0**					
t_{10}*	1	0	0	0	Fail
t_{11}*	0	1	0	0	Pass
t_{12}*	0	0	1	0	Fail
t_{13}*	0	0	0	1	Fail
result – (-, 0, -, -)					
for $t4$— **1 0 1 2**					
t_{14}*	2	0	1	2	Fail
t_{15}*	1	1	1	2	Pass
t_{16}*	1	0	2	2	Fail
t_{17}*	1	0	1	0	Fail
result – (-, 0, -, -)					
for $t7$— **2 0 2 1**					
t_{18}*	0	0	2	1	Fail
t_{19}*	2	1	2	1	Pass
t_{20}*	2	0	0	1	Fail
t_{21}*	2	0	2	2	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 stage. For example, when we identify the MFS of t_1 in Table 2, the schema (0, 1, -, -) contained in the extra passing test case t_{11} – (0, 0, 1, 0) has already been appeared in the passing test case t_2 – (0, 1, 1, 1). In another word, if we firstly isolate the MFS of t_1, then the t_2 is not the good choice as it doesn't covered as many as possible 2-value schemas, say, (1, 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 according to 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 may suffer from the *masking effects* [8], as failures caused by the already identified MFS can prevent the test case from normally checking (e.g., failures that can trigger an unexpected halt of the execution), as a result some schemas in these test cases that are supposed to be examined will actually skip the testing. Take the example in Table 2, after identifying the MFS – (-, 0, -, -) of t_1, we should not generate the test case t_4 and t_7. This because they also contain the identified MFS (-, 0, -, -), which will result in them failing as expected. Surely the expected failure caused by MFS (-, 0, -, -) makes t_4 and t_7 are superfluous for error-detection, and worse more some other schemas in t_4 or t_7 may be masked, as those schemas can potentially trigger other failures but will not be observed. And since we should not generate t_4 and t_7, then the additional test cases (t_{14} to t_{21})generated for identified the MFS in t_4 and t_7 are also not necessary.

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

3.2 New framework

To handle such deficiencies in traditional CT, we propose a new CT generation-localization framework. Our new framework aims at enhancing 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 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 module, 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.

The identify module works almost the same way as traditional independent MFS identify process, i.e., repeats generating and executing additional test cases until it can get enough information to diagnose 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 module have contributed 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 had not been

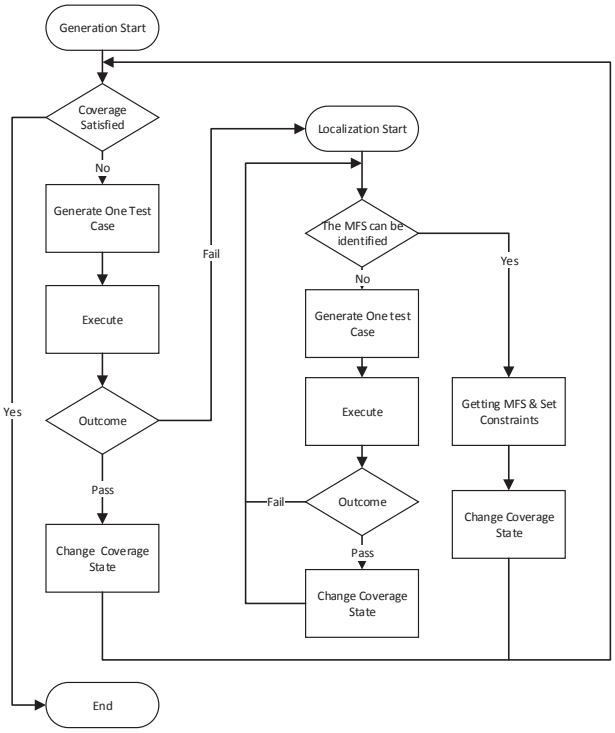

Figure 2: New Framework of CT

covered before. And when the MFS is found at the end of this module, we will first set them as forbidden schemas that latter generated test cases should not contain them (Otherwise the test case must fail and it cannot contribute to more coverage), second all the t-value schemas that are *related* to these MFS will be set as covered. Here the *related* indicates the following three types of t-value schemas:

First, the MFS **themselves**. Note that we haven't change the coverage state after the generated test case fails (both for the generation and identify module), so these MFS will never be covered as they always appear in these failing test cases.

Second, the schemas that are the **parent-schemas** of these MFS. By definition of the parent-schemas (Definition 3), we can find if the test case contain the parent-schemas, it must also contain all its sub-schemas. So every test case contain the parent-schemas of the MFS must fail after execution. As a result, they will never be covered as we don't change coverage state for failing test cases.

Third, those **implicit forbidden** schemas, which was first introduced in [3]. This type of schemas are caused by the conjunction of multiple MFS. For example, for a SUT with three parameters, and each parameter has two values, i.e., SUT(2, 2, 2). If there are two MFS for this SUT, which are (1, -, 1) and (0, 1, -). Then the schema (-, 1, 1) is the implicit forbidden schema. This is because for any test case that contain this schema, it must contain either (1, -, 1) or (0, 1, -). As a result, (-, 1, 1) will never be covered as all the test cases contain this schema will fail and so that we will not change the coverage state. In fact, by Definition 4, they can be deemed as faulty combinations.

As we all know, the terminating condition of the CT framework is to cover all the t-value schemas. Then since the three types of schemas will never be covered in our new CT framework, we must force to set them as covered after the execution of the identify module, so that the overall process can stop.

More details of these two important parts are as follows:

1) *Generation* : We adopt the one-test-case-one-time method as the basic skeleton of the generation process. And as discussed in section 3.1, 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 how to generate one test case 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: $value \leftarrow select(T_{uncovered}, Param, factor)$
6: **while not** $CheckCS(value, S_{MFS})$ **do**
7: $value \leftarrow repick(T_{uncovered}, Param, factor)$
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 [1] this criteria may be the test case which contain the most uncovered schemas. A candidate test case in this set is constructed by assigning specific values to each parameter in this test case. It is noted that we didn't specify that in which order these factors should be assigned, as this varies with different One-test-one-time generation methods. For each parameter under assignment (line 4), the value that is selected must satisfy two requirements: first, it should ensure that the test case under construction could cover as many uncovered schemas as possible (line 5); second, it must ensure that the test case under construction should **not** contain any MFS (line 6 - 8).

The first requirement is usually fulfilled by some heuristic selection, e.g., to choose the value for the parameter that is contained in the most uncovered schemas [1]. To fulfill the second requirement, a constraint satisfaction modeling is needed. A general model is as following:

$$X = P_1, P_2, P_3, P_n \quad (1)$$

$$D = D_1, D_2, ...D_n \quad (2)$$

$$C = C_{assignment}, C_{MFS} \quad (3)$$

In this formula, X is the parameters in the SUT and D is a set of the respective domains of values that each each parameter can take. C is the set of constraints. Then this model evaluates that whether a test case can be found, i.e., each parameter P_i takes a specific value D_i, so that it will not violate any constraint in C. For the constraints in C, $C_{assignment}$ indicates these parameters that have been assigned values. For example, if we have assigned parameter P_i with value d_i, and P_j with d_j, then this constraint will be formulated as $(P_i = d_i \;\&\&\; P_j == d_j)$. C_{MFS} modeled those identified MFS will be modeled as forbidden tuples of parameter values. For example, if (- 1 - 0) is the MFS, it will be transformed as forbidden rule $\neg(P_2 = 1 \;\&\&\; P_4 == 0)$.

So when using this model, we can check whether a value should be assigned to the current parameter(line 6) by putting it into the $C_{assignment}$. Note that the constraints checking part of our algorithm does not aim to optimising for the performance like running-time, iteration number or the like. Some study [4], 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 should also be adjusted to adapt to the new CT framework. From Figure 2, we can find some part of this process to isolate the MFS is similar to that of the *generation* module, i.e., they all need to repeat generating test cases until reach some criteria. As for the additional test cases generated in the isolation process, we should also take care that it should not contain the previously isolated MFS. To achieve this goal, the constraints checking process is also needed like Algorithm 1. Another point that needs to be noted is that the additional test cases generated in the isolation process can also contribute the coverage. As the overall testing process aims to cover all the t-value schemas, so if those additional test cases can cover more uncovered t-value schemas, the overall testing process can stop earlier. As a result, the overall test cases generated can be reduced. Based on the two points, the additional test cases generation in the CT isolation should be refined as in Algorithm 2.

We can observe that this algorithm is very similar to Algorithm 1. This can be easily understood, as the the target of Algorithm 2 is also the approach that can make the isolation to generate more uncovered tuples as well as to not contain the previous isolated MFS. Two different parts are:

This is mainly based on the fact that the machinaim of MFS isoaltion is *comparing*, that is, to compare the different schemas between the passing and failing test cases. And the fixed part is refer to the common part that is validated to error-irrelevant or not.

After the MFS are identified, some related schemas should be set as covered, i.e., these three types of schemas as we discussed before. The algorithm that seeks to handling these three types of schemas is listed in Algorithm 3.

In this algorithm, we firstly append the newly isolated MFS into the global MFS set (line 1 - 3), so that we can use them in the following generation and isolation processes. Then for each newly isolated MFS, we will set them as covered, i.e., remove them from the uncovered set, if they are t-degree schemas (line 4 - 7). This is the first type of schemas –*themselves*. For each t-degree parent-schema of

Algorithm 2 Test Case generation in isolation process

Input: $f_{original}$ ▷ original failing test case
S_{MFS} ▷ previously identified MFS
s_{fixed} ▷ fixed part that should not be changed
$T_{uncovered}$ ▷ the schemas that are still uncovered
$Param$ ▷ values set that each option can take
Output: t_{new} ▷ the regenerate test case

1: $Candidate_{test} \leftarrow emptySet$
2: **while** $Candidate_{test}.size$ is not satisfied **do**
3: $test \leftarrow emptyArray(Param.size)$
4: $s_{mutant} \leftarrow f_{original} - s_{fixed}$
5: **for each** $factor \in s_{mutant}$ **do**
6: $value \leftarrow select(T_{uncovered}, Param, factor)$
7: **while not** CheckCS(value, S_{MFS}) || value == GetValue(factor, $f_{original}$) **do**
8: $value \leftarrow repick(T_{uncovered}, Param, factor)$
9: **end while**
10: **end for**
11: $Candidate_{test}.append(test)$
12: **end while**
13: $best \leftarrow select(Candidate_{test})$
14: **return** *best*

Algorithm 3 Changing coverage after identification of MFS

Input: $S_{isolated}$ ▷ currently identified MFS
S_{MFS} ▷ previously identified MFS
$T_{uncovered}$ ▷ the schemas that are still uncovered

1: **for each** $s \in S_{isolated}$ **do**
2: $S_{MFS}.append(s)$
3: **end for**
4: **for each** $s \in S_{isolated}$ **do**
5: **if** s is t-degree schema **then**
6: $T_{uncovered}.remove(s)$
7: **end if**
8: **for each** s_p is parent-schema of s **do**
9: **if** s_p is t-degree schema **then**
10: $T_{uncovered}.remove(s_p)$
11: **end if**
12: **end for**
13: **end for**
14: **for each** $t \in T_{uncovered}$ **do**
15: **if not** $CheckCS(t, S_{MFS})$ **then**
16: $T_{uncovered}.remove(t)$
17: **end if**
18: **end for**

these newly isolated MFS, we will also remove them from the uncovered set (line 8 - 12), as they are the second type of schemas – *parent-schemas*. The last type, i.e., *implicit forbidden schemas*, is the toughest one. To remove them, we need to search through each potential schema in the uncovered schemas set (line 14), and judge if it is the implicit forbidden schema (line 16) by sat checking. This checking process is the same as we discussed in the generation and isolation process.

With this newly framework, when we re-consider the example in Table 2 in section 3.1, we can get the following result listed in Table 3.

Table 3: newly generation-isolation life-circle

generation					isolation				
t_1*	0	0	0	0	Fail				
						t_2*	1	0	0	Pass
						t_3*	0	1	0	Fail
						t_4*	0	0	1	Fail
						MFS: $(0, -, -)$				
t_5*	1	1	1		Pass					
t_6*	1	0	1		Pass					
t_7*	1	1	0		Pass					

This table consists of two main columns, where the left indicates the generation part while the right column indicates the isolation process.

Up to now, all the 2-degree schemas are covered, in detail, (1 1 -) (1 - 1) (- 1 1) are covered by passing test case (1 1 1), (1 0 -) (1 - 0) (- 0 0) are covered by passing test case (1 0 0), (- 0 1) and (- 1 0) are covered by test case (1 0 1) and (1 1 0) respectively. Those schemas related to (0 - -) such as (0 - 1), (0 0 -) will not be covered as the (0 - -) is the MFS. All the generated 7 test cases are not duplicated with each other (all marked *), we can find the overall generated test case are one less than the traditional approaches in Table 2. Another advances of our approach is that our approach provide a stronger covering criteria than traditional covering array.

4. EMPIRICAL STUDIES

4.1 Compare with Traditional First-Gen-Latter-Identify

4.1.1 Study setup

4.1.2 Result and discussion

4.2 Compare with the FDA-CIT

4.2.1 Study setup

4.2.2 Result and discussion

4.3 Comparing with Error locating Array

Erray locating array [5, 7] is a well-designed set of test cases, such that it can support not only failure detection, but also the identification for the MFS of the failure.

4.3.1 Study setup

4.3.2 Result and discussion

4.4 Threats to validity

5. RELATED WORKS

6. CONCLUSIONS

7. 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] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 129–139. ACM, 2007.

[4] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *Software Engineering, IEEE Transactions on*, 34(5):633–650, 2008.

[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] 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.

[7] 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.

[8] C. Yilmaz, E. Dumlu, M. Cohen, and A. Porter. Reducing masking effects in combinatorialinteraction testing: A feedback drivenadaptive approach. *Software Engineering, IEEE Transactions on*, 40(1):43–66, Jan 2014.

[9] S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):19, 2013.