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

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

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

JiaXi Xu
School of Information Engineering
Nanjing Xiaozhuang University
China, 211171
xujiaxi@njxzc.edu.cn

Yan Wang
School of Information Engineering
Nanjing Xiaozhuang University
China, 211171
wangyan@njxzc.edu.cn

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 localize 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 invalid 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.

*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

Categories and Subject Descriptors

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

General Terms

Reliability, Verification

Keywords

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

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 because 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 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 [30]; 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 the 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. Specifically, 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 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. The second one is to compare with the Error Locating Array [20, 21], which is a well-designed set of test cases that can be used directly detect and localize the failure-inducing interactions. The results shows that in terms of test case generation and fault diagnosis, our approach can significantly reduced the overall needed test cases and as a result it can more quickly localize the root cause of the system under test.

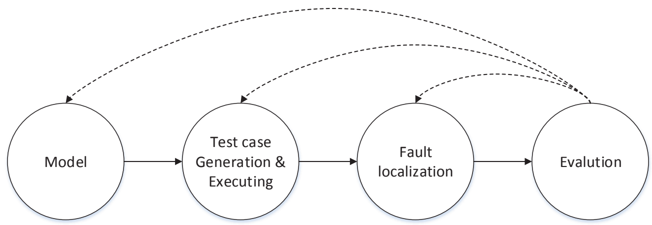

Figure 1: The life circle of CT

The main contributions of this paper:

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

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

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

The rest 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 can take the values from the finite set V_i (i = 1,2,..n). Some of the definitions below are originally defined in [22].

Definition 1. A *test case* of the SUT is an tuple 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-degree *schema* ($0 < k \leq n$) 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-degree *schema*, when k = n. Furthermore, if a test case contain a *schema*, i.e., every fixed value in the schema is in this test case, we say this test case *hits* the *schema*.

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 *parent-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 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 sub-schema of c is the *faulty schema* for F, we then call the schema c the *minimal failure-causing schema* [22] for F.

In fact, MFS are identical to the failure-inducing interactions we discussed previously. Figuring it out helps to focus on the root cause of a failure and thus 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 called the *covering array.* If some test cases in the covering array failed in 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 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 focus 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[4].

3) Others : This strategy differentiates from the previous two strategies at the point it does not first give completed test cases [18]. 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 Localize 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 $(v_1, v_2, ..., v_n)$, there can be at most $2^n - 1$ possible schemas can be the MFS. Hence, further fault diagnosis is desired, i.e., more test cases should be generated to localize the MFS.

A typical MFS localization 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
Additional test cases				
t_1	0	1	1	Pass
t_2	1	0	1	Fail
t_3	1	1	0	Fail

This localization 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 localization 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 are the most important works 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-localization 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-localization process

A typical traditional generation-localization 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 localize 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 three values. It first generated and executed the 2-way covering array (t_1 – t_9). Note that this covering array covered all the 2-way schemas for the SUT. And after finding that t_1, t_2, and t_7 failed during testing, it then respectively localized 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 localizing the MFS in this example SUT, we have generated 12 additional test cases(marked with star).

Table 2: traditional generation-localization 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 after all the test cases are executed. 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, 1, 0, 0) has already been appeared in the passing test case t_2 – (0, 1, 1, 1). In another word, if we firstly localize the MFS of t_1, then the t_2 is not the good choice as it doesn't covered as many as possible 2-degree 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* [29], 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-degree schemas, with t normally be assigned as 2 or 3. Then if the coverage currently is not satisfied, it will generate a new test case to cover as many uncovered combinations 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

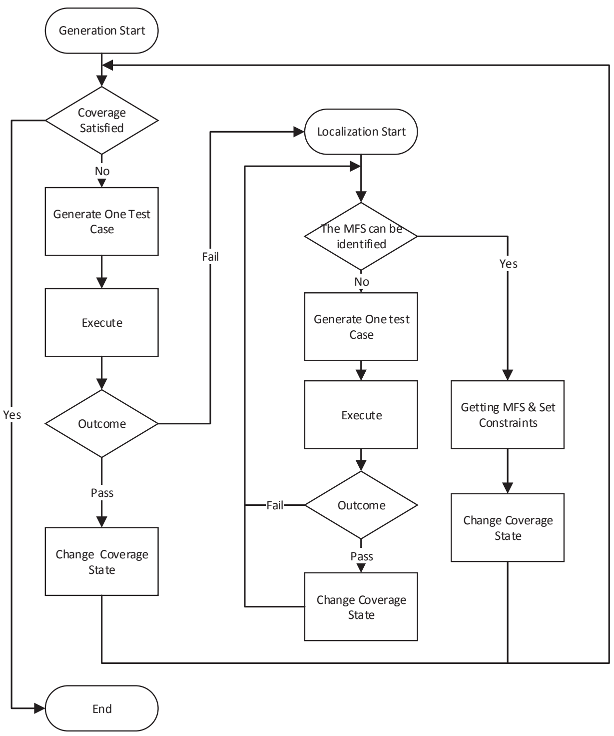

Figure 2: New Framework of CT

this newly generated test case. Otherwise if the test case fails, then we will start the MFS identify module, to localize the MFS in this failing test case. One point that needs to be noted is that if the test case fails, we will not directly 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 localize 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 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-degree schemas that are *related* to these MFS will be set as covered. Here the *related* indicates the following three types of t-degree 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 [6]. 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-degree 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 [6, 7], 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: $Params$ ▷ Parameters for the SUT
$Values$ ▷ Corresponding values for each parameter
S_{MFS} ▷ the set of MFS that currently localized
$T_{uncovered}$ ▷ the schemas that are still uncovered
Output: $best$ ▷ the generated test case

1: $Candidate_{test} \leftarrow emptySet$
2: **while** $Candidate_{test}$ is full **do**
3: $test \leftarrow emptyTuple(Params.size)$
4: **for each** $p \in Params$ **do**
5: $v \leftarrow select(T_{uncovered}, Values, p)$
6: **while not** $isSatisfied(v, S_{MFS})$ **do**
7: $v \leftarrow repick(T_{uncovered}, Values, p)$
8: **end while**
9: $test.set(p, v)$
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 [3] 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 [3]. To fulfill the second requirement, a constraint satisfaction modeling is needed. A general model is as following:

$$X = P_1, P_2, P_3, P_n \tag{1}$$

$$D = V_1, V_2, ...V_n \tag{2}$$

$$C = C_{MFS}, C_{assignment} \tag{3}$$

In this formula, X is the parameters in the SUT and V is a set of the respective domains of values that 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 v_i ($v_i \in V_i$), so that it will not violate any constraint in C. For the constraints in C, C_{MFS} indicates those identified MFS that should not be contained in the test case. For example, if (- 1 - 0) is the MFS, it will be transformed as forbidden rule $\urcorner(P_2 = 1$ && $P_4 = 0)$. $C_{assignment}$ indicates these parameters that have been assigned values. For example, if we have assigned parameter P_i with value v_i, and P_j with d_j, then this constraint will be formulated as ($P_i = v_i$ && $P_j = v_j$). Note that if we can not find a test case that fulfill this constraint satisfaction formula, it means that the values that have been assigned to those parameters, i.e., $C_{assignment}$, are invalid.

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 [7], 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) *Localization* : The localization process should also be adjusted to adapt to the new CT framework. From Figure 2, we can find some part of this process to localize 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 localization process, we should also take care that it should not contain the previously localized 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 localization process can also contribute the coverage. As the overall testing process aims to cover all the t-degree schemas, so if those additional test cases can cover more uncovered t-degree 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 localization should be refined as in Algorithm 2.

Algorithm 2 Test Case generation in localization process

Input: $Params$ ▷ Parameters for the SUT
$Values$ ▷ Corresponding values for each parameter
$f_{original}$ ▷ 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
Output: $best$ ▷ the generated additional test case

1: $Candidate_{test} \leftarrow emptySet$
2: **while** $Candidate_{test}$ is not full **do**
3: $\quad test \leftarrow emptyTuple(Params.size)$
4: $\quad s_{mutant} \leftarrow f_{original} - s_{fixed}$
5: $\quad$ **for each** $p \in s_{mutant}$ **do**
6: $\quad\quad v \leftarrow select(T_{uncovered}, Values, p)$
7: $\quad\quad$ **while** (**not** isSatisfied(v, S_{MFS}) || $f_{original}$.contain(p,v))**do**
8: $\quad\quad\quad v \leftarrow repick(T_{uncovered}, Values, p)$
9: $\quad\quad$ **end while**
10: $\quad$ **end for**
11: $\quad Candidate_{test}.append(test)$
12: **end while**
13: $best \leftarrow select(Candidate_{test})$
14: **return** $best$

We can observe that this algorithm is very similar to Algorithm 1, except that this algorithm introduce the variables $f_{original}$ and s_{fixed}. These two variables are important to MFS localization. Generally, the target of the MFS localization process is to distinguish the failure-inducing schemas from those error-irrelevant schemas in a failing test case. For this, the MFS localization process need to generate and execute additional test cases to compare to original failing test case $f_{original}$. The additional test case must contain some fixed schema s_{fixed} in the $f_{original}$, and other part of the additional test case must be different from $f_{original}$ (line 4). By doing this, this localization process can check whether the *fixed* are failure-inducing or not. For example, in Table 1, the original failing test case is (1, 1, 1) and the fixed part for additional test case t_1 (0, 1, 1) is (-, 1, 1). After the t_1 passed during testing, we can get that the fixed schema (-, 1, 1) should not be the failure-inducing schema.

Traditional MFS localization process just need to ensure that the *mutant* part have different values from original failing test cases. We augmented this by selecting values that can cover as more uncovered schemas as possible (line 6) and to ensure the test case does not contain some identified MFS (line 7 - 8).

After the MFS are identified, some related t-degree schemas, i.e., *MFS themselves*, *parent-schemas* and *implicit forbidden schemas*, should be set as covered to make the overall C-T process stoppable. The algorithm that seeks to handling these three types of schemas is listed in Algorithm 3.

In this algorithm, we firstly append the newly localized MFS into the global MFS set (line 1 - 3), so that we can use them in the following generation and localization processes. Then for each newly localized 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 these newly localized 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*

Algorithm 3 Changing coverage after identification of MFS

Input: $S_{localized}$ ▷ currently localized MFS
S_{MFS} ▷ previously identified MFS
$T_{uncovered}$ ▷ the schemas that are still uncovered
Output: *void* ▷ do not need any output

```
1: for each s ∈ S_localized do
2:     S_MFS.append(s)
3: end for
4: for each s ∈ S_localized 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 ∈ T_uncovered do
15:     if not isSatisfied(t, S_MFS) then
16:         T_uncovered.remove(t)
17:     end if
18: end for
```

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 constraints checking. This checking process is the same as we discussed in the Algorithm 1 and Algorithm 2.

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-localization life-circle

Generation						*Localization*					
t_1	0	0	0	0	Fail						
						t_2*	1	0	0	0	Fail
						t_3*	0	1	0	0	Pass
						t_4*	0	0	1	0	Fail
						t_5*	0	0	0	1	Fail
						MFS: $(-, 0, -, -)$					
t_6	0	1	1	1	Pass						
t_7	0	2	2	2	Pass						
t_8	1	1	1	2	Pass						
t_9	1	1	2	0	Pass						
t_{10}	1	2	0	1	Pass						
t_{11}	2	1	2	1	Pass						
t_{12}	2	1	0	2	Pass						
t_{13}	2	2	1	0	Pass						

This table consists of two main columns, where the left indicates the generation part while the right column indicates the localization process. We can find that after localizing the MFS (-, 0, -, -) for t_1. The following test cases (t_6 to t_{13}) will not contain this schema. Correspondingly, all the 2-degree schemas that are related to this schema, e.g. (0, 0, -, -), (-, 0, 1, -), etc, will also not appear in the following test cases. Additionally, the passing test case t_3 generated in the localization process cover 6 *2*-degree schemas, i.e., (0, 1, -, -), (0, -, 0, -), (0, -, -, 0), (-, 1, 0, -), (-, 1, -, 0), and (-, -, 0, 0) respectively, so that it is not necessary to generate more test cases to cover them. Above all, with the new CT framework, the overall generated test case are 8 less than that of the traditional CT framework in Table 2.

Note that this example only lists the condition of a single MFS, under which some *parent-schemas* or *themselves* will do not need to be covered. When there are multiple MFS, additional *implicit forbidden* schemas will be computed and set to be covered.

4. EMPIRICAL STUDIES

To evaluate the effectiveness and efficiency of our new CT framework, we conducted a series of empirical experiments on several open-source software.

4.1 Subject programs

The subject programs used in our experiments are five open-source software as listed in Table 4. Column "Subjects" indicates the specific software. Column "Version" indicates the specific version that are used in the following experiments. Column "LOC" shows the number of source code lines for each software. Column "Faults" presents the fault ID, which can used as the index to fetch the original fault description at each bug tracker for that software. Column "Lan" shows the programming language for each software (Only the main programming language are shown).

Table 4: Subject programs

Subjects	Version	LOC	Faults	Lan
Tomcat	7.0.40	296138	#55905	java
Hsqldb	2.0rc8	139425	#981	Java
Gcc	4.7.2	2231564	#55459	c
Jflex	1.4.1	10040	#87	Java
Tcas	v1	173	#Seed	c

In these software, Tomcat is a web server for java servlet, Hsqldb is a pure-java relational database engine, Gcc is the programming language compiler, Jflex is a lexical analyzer generator, and Tcas is a module of an aircraft collision avoidance system. We take these software as subjects program because their behaviours are influenced by various combinations of configuration options or inputs. For example, one component *connector* of Tomcat is influenced by more than 151 attributes [14]. For program Tcas, although with a relatively small size (only 173 lines), it also has 12 parameters with their values ranged from 2 to 10. As a result, the overall input space for Tcas has reached to 1036800 [27, 16].

As the target of our empirical studies is to compare the ability of fault defecting between our approach with traditional ones. We firstly must know these faults and their corresponding MFS in prior, such that we can determine whether the schemas identified by those approaches are accurate or not. For this, We looked through the bug tracker forum of each software and focused on the bugs which are caused by the options combination. Then for each such bug, we will derive its MFS by analysing the bug description report and the associated test file which can reproduce the bug. For Tcas, as it does not contain any fault for the original source file, we took an mutation version for that file with injected fault. The mutation was the same as that in [16], which is usually as a experiment object for the fault defeating studies.

4.1.1 Specific inputs models

To apply the CT on the prepared software, we need to firstly model their input parameters. As we discussed before, the whole configuration options is extremely large so that we cannot include all of them in our model in consideration of the experiment time and computing resource. Instead, a moderate small set of these configuration options will be selected. It is noted that the options that are caused the specific fault in Table 4 will be included as so the test cases generated by CT will detect that fault. Additional options are selected to make some noise for the MFS identifying approach. These options are selected by random. The inputs model as well as the corresponding MFS (degree) are listed as Table 5.

Table 5: inputs model

Subjects	Inputs	MFS
Tomcat	$2^8 \times 3^1 \times 4^1$	1(1) 2(2)
Hsqldb	$2^9 \times 3^2 \times 4^1$	3(2)
Gcc	$2^9 \times 6^1$	3(4)
Jflex	$2^{10} \times 3^2 \times 4^1$	2(1)
Tcas	$2^7 \times 3^2 \times 4^1 \times 10^2$	9(16) 10(8) 11(16) 12(8)

In this table, Column "inputs" depicts the input model for each version of the software, presented in the abbreviated form $\#values^{\#number\ of\ parameters} \times ...$, e.g., $2^9 \times 3^2 \times 4^1$ indicates the software has 9 parameters that can take 2 values, 2 parameters can take 3 values, and only one parameter that can take 4 values. Column "MFS" shows the degrees of each MFS and the corresponding number which are listed in the parentheses.

4.2 Compare with Traditional First-Generating-Then-Identifying approach

After preparing the subjects software, next we will construct the experiment that can evaluate the efficiency and effectiveness of our approach. To this aim, we need to compare our framework with the traditional CT framework to see if our new framework have any advantage.

The covering array generating algorithm used in the experiment are AETG [3], for this is the most common one-test-case-one-time generation algorithm. And the MFS identifying algorithm is the OFOT [22] as we discussed before. The constraints handling solver is a java SAT solver – SAT4j [17].

4.2.1 Study setup

In this experiment, we focus on three coverage criteria, i.e., 2-way , 3-way and 4-way, respectively. It is known that the generated test cases vary for different runs of AETG algorithm. So to avoid the biases of randomness, we conduct each experiment 30 times and evaluate the results. In other word, for each subject software, we will repeatedly execute traditional approach and our approach 30 times to detect and localize the MFS.

To evaluate the results of the two approaches, one metric is the cost, i.e., the overall test cases that each approach needs. Apart from this, another important metric is the quality of their identified MFS. For this, we used standard metrics: *precise* and *recall*, which are defined as following:

$$precise = \frac{\#the\ num\ of\ correctly\ identified\ MFS}{\#the\ num\ of\ all\ the\ identified\ schemas}$$

and

$$recall = \frac{\#the\ num\ of\ correctly\ identified\ MFS}{\#the\ num\ of\ all\ the\ real\ MFS}$$

Precise shows the accurateness of the identified schemas when comparing to the real MFS. *Recall* measures how well the real MFS are detected and localized. The combination of them is F-measure, which is

$$F - measure = \frac{2 \times precise \times recall}{precise + recall}$$

4.2.2 Result and discussion

The result is listed in Table 6. In Column 'Method', *elda* indicates our new CT framework and *fglt* indicates the traditional first-gen-then-identify approach. The results of three coveing criteria, i.e., 2-way, 3-way, and 4-way are shown in three main columns. In each of them, the overall test cases (size), precise, recall, and f-measure and listed.

One observation from this table is that the overall test cases generated by our approach are far less than that of the traditional approach. In fact, except for subject *tcas*, our approach reduced about dozens of test cases for 2-way coverage, and hundreds test cases for 3-way and 4-way coverage. For *tcas*, however, as the MFS are hard to detect (all of them have degrees than 9), so both two approaches nearly do not trigger errors (see the metric *recall*). Under this condition, both the two approaches will be transferred to a normal covering array.

As for the quality of the identified MFS, we find that there is no apparent gap between them. For example, there are 9 cases under which our approach performed better than traditional one (marked in bold). But among these cases, the maximal gap are 0.27 (2-way for Tomcat), and the average gap are around 0.1, which is trivial.

The reason of the similarity between the quality of these two approaches is that both of them have advantages and disadvantages. Specifically, our approach can reduce the impacts when a test case contain multiple MFS. Our previous study showed that multiple MFS in a test case can reduce the accurateness of the MFS identifying algorithms [25]. As a result, our approach can improve the quality of the identified schemas. But as a side-effect, if the schemas identified at the early iteration of our approach are not correct, it will significantly impact the following iteration. This is because we will compute the coverage and constraints based on previous identified MFS. It was the other way around for traditional approach. Traditional one suffers from impacts when a test case contain multiple MFS, but correspondingly, previous identified MFS has little influence on the traditional approach.

In a sum, our approach needs much less test cases than traditional first-generating-then-identifying approach, and there is no decline in the quality of the identified MFS when comparing with traditional approach.

4.3 Comparing with Error locating Array

Error locating array[9, 21] is a well-designed set of test cases, such that it can support not only failure detection, but also the localization of the MFS of the failure. It is known

Table 6: Compare with traditional approach

Subject	Method	2-way				3-way				4-way			
		Size	Precise	Recall	F-measure	Size	Precise	Recall	F-measure	Size	Precise	Recall	F-measure
Tomcat	elda	42.53	1	1	**1**	64.67	0.97	0.96	**0.96**	114.23	0.93	0.91	**0.92**
	fglt	114.33	0.73	0.74	0.73	289.97	0.75	1	0.86	676.83	0.75	1	0.86
Hsqldb	elda	27.3	0.67	0.4	**0.49**	63.07	0.37	0.37	0.37	144.93	0.37	0.37	0.37
	fglt	26.43	0.55	0.3	0.38	80.77	0.5	0.5	0.5	233.13	0.5	0.5	0.5
Gcc	elda	32.73	0.47	0.28	**0.34**	72.5	0.46	0.37	**0.40**	130.67	0.58	0.48	**0.52**
	fglt	34.17	0.33	0.18	0.23	114.5	0.36	0.43	0.39	243.37	0.33	0.5	0.40
Jflex	elda	29.97	1	1	1	63.3	1	1	1	159.77	1	1	1
	fglt	48.27	1	1	1	187.93	1	1	1	580.23	1	1	1
Tcas	elda	108.63	0	0	0	426.77	0.03	0.0007	**0.001**	1638.23	0.07	0.001	**0.0027**
	fglt	109	0	0	0	426.83	0	0	0	1638.57	0.03	0.001	0.0026

that only with a covering array sometimes is not sufficient to localize the MFS, thus additional test cases are needed. Martínez et al.[20] has proved that a $(t+d)$-way covering array can localize all the MFS with the number of them no more than d, and degrees no more than t. After executing all the test cases in the $t+d$-way covering array, the MFS can be obtained by keeping those t-degree or less than t-degree schemas that only appear in the failing test cases in the covering array. So with the number d and degrees t known in prior, a $(t+d)$-way covering array is a *Error Locating Array (ELA)*.

To compare our approach with this CT-based array is meaningful, as both this two approaches have the same target. The relationship between our approach with the Error locating array can be deemed as the dynamic and static. In detail, our approach dynamically detect and localize the MFS in the SUT, i.e., the test cases generated by our approach are changed according to the specific MFS. On the contrary, ELA just generate a static covering array, and it can support MFS identification if the number and degrees of these schemas are known in prior.

4.3.1 Study setup

As the results of our approach have been already collected in the Section 4.2, so in this section will just apply ELA to identify the MFS of the 5 prepared software in Table 4. It is noted that the conclusion that a t+d-way covering array is an ELA is based on that there must exist *safe* values for each parameter of the SUT. A *safe* value is the parameter value that is not the part of these MFS. In our experiment, all the five subject programs satisfies this condition. Based on this, we then applied the ELA to generate appropriate covering arrays for the each subject program and recorded the MFS they identified and as well as the overall test cases it generated. The covering array generation algorithm we adopted in this experiment is also AETG [3], and similar as the previous experiment, will repeat this experiment 30 times.

4.3.2 Result and discussion

The overall test cases and the quality of the identified MFS are listed in Table 7. We can firstly observe that this approach needs much more test cases than the two approaches we discussed before. This is as expected, as this approach needs to generate a higher-way covering array than previous two approaches. Apart from the high cost, this approach correctly identified all the real MFS. The accurateness has been proved in [20, 21]. Note that this perfect MFS identification result is based on the fact that it knows the number and degrees of the MFS, which is usually not available in practice.

Table 7: Compare with Error Locating Array

Subject	Size	Precise	Recall	F-measure
Tomcat	210.8	1	1	1
Hsqldb	333.8	1	1	1
Gcc	860.4	1	1	1
Jflex	49.1	1	1	1
Tcas	460800	1	1	1

As both the cost (number of test cases) and the quality of the identified schemas are important in practice. We combine the two metrics (*size* and *f-measure*) into a single one, by diving *f-measure* by *size*. The normalized result of this combination metric is listed in Fig.3. In this figure, *elda* and *fglt* represent our new approach and traditional first-generating-then-identifying approach. *Ela* indicates the error locating array approach.

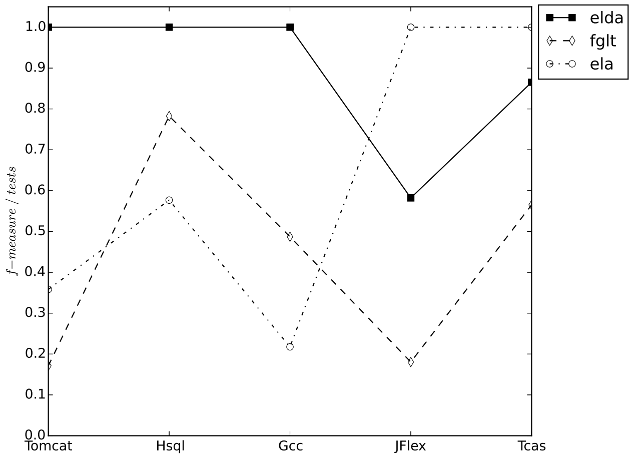

Figure 3: performance comparison

We can learn that *elda* performs better than *fglt* at all the five subjects. For *ela*, our approach performs better than it at three subjects *Tomcat*, *Hsqldb*, and *Gcc*. The reason that our approach does not perform as well as *ela* at subject *JFlex* is that the MFS for that object is a single 2-degree schema (see Table 5), under which ela just needs a 3-way covering array. For subject *Tcas*, with high-degree MFS as we discussed before, our approach is hard to trigger an errors with only *2-way*, *3-way*, and *4-way* covering array. As a result, our approach can hardly identify the MFS. Apart from this two exception, our approach have an significant advantage over the ELA approach.

To summarize, ELA get the best at the quality of the MFS, but needs much more test cases than our approach. What's more, needing to know the number and degrees of the MFS in prior limiting the application of ELA in practice.

4.4 Threats to validity

There are several threats to validity in our empirical studies. First, our experiments are based on only 5 open-source software, more subject programs are desired to make the results more general. In fact, we must take comprehensive experiments on the programs with parameters and MFS under control, such that the conclusion of our experiment can reduce the impact caused by specific input space and specific degree or location of the MFS.

Second, many more generation algorithm and MFS localization algorithms are needed. In our empirical studies, we just used the AETG [3] as test cases generation strategy, and OFOT [22] as the MFS identification strategy. As the different generation and identification algorithms will significantly affect the performance our proposed CT framework, especially for the number of test cases. More studies for different test cases and MFS identification algorithms are desired to figure this impact out.

Third, these empirical studies are all on the deterministic condition, i.e., the output of the software are deterministic if their inputs are given. When the output are affected by random events such that we can not determine the output by only one-time execution of the test case, then both the traditional CT process and our new framework do not work. In such a case, to conduct a test case multiple times or introduce some probability in the framework will be of interest.

5. RELATED WORKS

Combinatorial testing has been an widely applied in practice [15], especially on domains like configuration testing [28, 8, 26, 11] and software inputs testing [3, 1, 13, 12]. A recent survey [23] comprehensively studied existing works in CT and classified those works into eight categories according to the testing procedure. Based on which, we can learn that test cases generation and fault diagnosis are two most important key part in CT studies.

Although CT has been proven to be effective at detecting and localizing the interaction failures in SUT, however, to directly applied them in practice can be inefficient and some times even not work at all. Some problems, e.g., constraints of parameters values in SUT [5, 7], masking effects of multiple failures[10, 29], dynamic requirement for the strength of covering array [11], will bring many troubles to CT process. To overcome these problems, some works made efforts to make CT more adaptive and flexible.

JieLi [19] augmented the MFS identifying algorithm by selecting one previous passing test cases for comparison, such that it can reduce some extra test cases when compared to another efficient MFS identifying algorithm [31]. S.Fouché et al., [11] introduced the notion of incremental covering. Different from traditional covering array, it did not need a fixed strength to guide the generation, instead, it can dynamically generate high-degree covering array based on existing low-degree covering array, which can support a flexible tradeoff between covering array strength and testing resources. Cohen [5, 7] studied the impacts of constraints for CT, and proposed an SAT-based approach that can handle those constraints. Bryce and Colbourn [2] proposed an one-test-case-one-time greedy technique to not only generate test cases to cover all the t-way interactions, but also prioritize them according their importance. E. Dumlu et al., [10] developed a feedback driven combinatorial testing approach that can assist traditional covering in avoiding these masking effects between multiple failures. Yilmaz [29] extended that work by refining the MFS diagnosing method in it. Additionally, Nie [24] constructed an adaptive combinatorial testing framework, which can dynamically adjusted the inputs model, strength t of covering array, and generation strategy during CT process.

Our work differs from them mainly at our work focus on combining two important techniques in CT, i.e., test cases generation and fault diagnosis, such that the overall cost of CT will be reduced and the identified MFS will be of higher quality.

6. CONCLUSIONS

Combinatorial testing is an effective testing technique at detecting and diagnosis the failure-inducing schemas in the SUT. Traditional CT works separately at cases generation and MFS localization. In this paper, we proposed a new CT framework that organised these two important testing stages as a whole, which allows for both generation and localization better share each other's information. As a result, our new CT framework can provide a more efficient testing than traditional CT.

Empirical studies were conducted on several open-source software. The results showed that with our new CT framework, there is an significant reducing on the number of generated test cases when compared to traditional first-generation-then-localization process, while there is no decline in the quality of the identified MFS. Further, when comparing with the ELA [21, 20], our approach also performed better, especially at the number of test cases.

As a future work, we need to extend our new CT framework with more test cases generation and MFS localization algorithms, to see the extent on which our new CT framework can enhance those different CT-based algorithms. Another interesting work is to combine our new CT framework with white-box testing techniques, so that it can provide more useful information for developers to debug the system.

7. REFERENCES

[1] M. N. Borazjany, L. Yu, Y. Lei, R. Kacker, and R. Kuhn. Combinatorial testing of acts: A case study. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 591–600. IEEE, 2012.

[2] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding

and constraints. *Information and Software Technology*, 48(10):960–970, 2006.

[3] 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.

[4] 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.

[5] M. B. Cohen, M. B. Dwyer, and J. Shi. Exploiting constraint solving history to construct interaction test suites. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007.*, pages 121–132. IEEE, 2007.

[6] 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.

[7] 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.

[8] M. B. Cohen, J. Snyder, and G. Rothermel. Testing across configurations: implications for combinatorial testing. *ACM SIGSOFT Software Engineering Notes*, 31(6):1–9, 2006.

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

[10] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter. Feedback driven adaptive combinatorial testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 243–253. ACM, 2011.

[11] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 177–188. ACM, 2009.

[12] B. Garn and D. E. Simos. Eris: A tool for combinatorial testing of the linux system call interface. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pages 58–67. IEEE, 2014.

[13] L. S. G. Ghandehari, M. N. Bourazjany, Y. Lei, R. N. Kacker, and D. R. Kuhn. Applying combinatorial testing to the siemens suite. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 362–371. IEEE, 2013.

[14] K. Kolinko. The HTTP Connector. `http://tomcat.apache.org/tomcat-7.0-doc/config/http.html`, 2014. [Online; accessed 3-Nov-2014].

[15] D. R. Kuhn, R. N. Kacker, and Y. Lei. Practical combinatorial testing. *NIST Special Publication*, 800:142, 2010.

[16] D. R. Kuhn and V. Okun. Pseudo-exhaustive testing for software. In *Software Engineering Workshop, 2006. SEW'06. 30th Annual IEEE/NASA*, pages 153–158. IEEE, 2006.

[17] D. Le Berre, A. Parrain, et al. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[18] 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.

[19] J. Li, C. Nie, and Y. Lei. Improved delta debugging based on combinatorial testing. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 102–105. IEEE, 2012.

[20] C. Martínez, L. Moura, D. Panario, and B. Stevens. Algorithms to locate errors using covering arrays. In *LATIN 2008: Theoretical Informatics*, pages 504–519. Springer, 2008.

[21] 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.

[22] 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.

[23] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11, 2011.

[24] C. Nie, H. Leung, and K.-Y. Cai. Adaptive combinatorial testing. In *Quality Software (QSIC), 2013 13th International Conference on*, pages 284–287. IEEE, 2013.

[25] 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.

[26] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 75–86. ACM, 2008.

[27] K. Shakya, T. Xie, N. Li, Y. Lei, R. Kacker, and R. Kuhn. Isolating failure-inducing combinations in combinatorial testing using test augmentation and classification. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 620–623. IEEE, 2012.

[28] C. Yilmaz, M. B. Cohen, and A. A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on*, 32(1):20–34, 2006.

[29] 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.

[30] 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.
[31] 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.