An interleaving approach to combinatorial testing and failure-inducing interaction identification*

Xintao Niu and 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
ylei@cse.uta.edu

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

Xiaoyin Wang
Department of Computer Science
University of Texas at San Antonio
Xiaoyin.Wang@utsa.edu

ABSTRACT

Combinatorial testing(CT) seeks to detect 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 set of test cases to cover each possible interaction and then to identify the failure-inducing interaction after a 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 desire to identify the root cause of failures before all the needed test cases are generated and executed 2) the early identified failure-inducing interactions can guide the remaining test cases generation, such that many unnecessary and invalid test cases can be avoided. For these reasons, we propose a novel CT framework that allows both generation and identification process to interact with each other. As a result, both generation and identification stages will be done more effectively and efficiently. We conducted a series of empirical studies on several open-source software, the results of which show that our framework can identify the failure-inducing interactions more quickly than traditional approaches, while requiring fewer 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 becoming more and more complex. To test such software is challenging, as the candidate factors that can influence the system's behaviour, e.g., configuration options, system inputs, message events, are enormous. Even worse, the interactions between these factors can also crash the system, e.g., the incompatibility problems. In consideration of the scale of the real industrial software, to test all the possible interactions of all the factors (we call them the interaction space) is not feasible, and even if it is possible, it is not wise to test all the interactions because most of them do not provide any useful information.

Many empirical studies show that, in real software systems, the effective interaction space, i.e., targeting fault detection, makes up only a small proportion of the overall interaction space [17, 18]. What's more, the number of factors involved in these effective interactions is relatively small, of which 4 to 6 is usually the upper bounds[17]. With this observation, applying CT in practice is appealing, as it is proven to be effective to detect the interaction faults in the system.

A typical CT life-cycle is shown in Figure 1, which contains 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, that could affect the system's behavior. Further effort is needed to figure out

the constraints and dependencies between each factor and corresponding values for valid testing. After the modeling stage, a set of test cases should be generated and executed to expose the potential faults in the system. In CT, each test case is a set of assignments of all the factors in the test model. 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 set of test cases to achieve some specific coverage. The third testing stage in this cycle is the fault localization, which is responsible for identifying the failure-inducing interactions. To characterize the failure-inducing interactions of corresponding factors and values is important for future bug fixing, as it will reduce the suspicious code scope to be inspected. The last testing stage of CT is evaluation. In this stage, testers will assess the quality of the previously conducted testing tasks. If the assessment result shows that the previous testing process does not fulfil the testing requirement, some testing stages should be improved, and sometimes, may even need to be re-conducted.

Although this conventional CT framework is simple and straightforward, in terms of the test cases generation and fault localization stages, we conjecture that first-generation-then-identification is not the proper choice in practice. The reasons are twofold. First, it is not realistic for testers to wait for all the needed test cases are generated before they can diagnose and fix the failures that have been detected [34]; 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 can be avoided. For this we get the key idea of this paper: *Generation and Fault Localization process should be integrated.*

Based on the idea, we propose a new CT framework, which instead of dividing the generation and identification into two independent stages, it integrates these two stages into one. Specifically, we first execute one or more tests until a failure is observed. Next we immediately turn to the fault localization stage, i.e., identify failure-inducing interactions for that failure. These failure-inducing interactions are used to update the currently coverage. In particular, interactions that are related to these failure-inducing interactions do not need to be covered in future executions. Then, we go back to perform regular combinatorial testing and continue.

We remodel the test cases generation and failure-inducing interactions identification modules to make them better adapt to this new framework. Specifically, for the generation part of our framework, we augment it by forbidding the appearance of test cases which contain the identified failure-inducing interactions. This is because those test cases contain a failure-inducing interaction will fail as expected, so that it makes no sense for the further failure detection. For the failure-inducing identification module, we augment it by making them to contribute more coverage. More specifically, we refine the additional test cases generation in this module, so that it can not only help to identify the failure-inducing interactions, but also cover as many uncovered interactions as possible. As a result, our new CT framework needs fewer test cases than traditional CT.

We conducted a series of empirical studies on several open-source software to evaluate our new framework. Specifically, we performed two comparisons studies. The first one is to compare our new framework with the traditional one, which first generates a complete set of test cases and then performs the fault localization. The second one is to compare our framework with the Feedback-driven CT [10, 33], which also adapts an iterative framework to generate test cases and identifying failure-inducing interactions, but to address the problem of inadequate testing. The results show that, in terms of test case generation and failure-inducing interactions identification, our approach can significantly reduce the overall needed test cases and as a result it can more quickly identify the failure-inducing interactions of the system under test. Additionally, when combining our approach with Feedback-driven CT, the results can be more effective than both approaches alone.

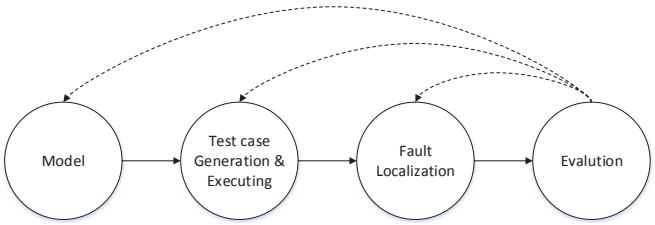

Figure 1: The life cycle of CT

The main contributions of this paper are as follows.

1. We propose a new CT framework which combines the test cases generation and fault localization more closely.
2. We augment the traditional CT test cases generation and failure-inducing interactions identification process to make them adapt to the new framework.
3. We perform a series of comparisons with traditional CT and Feedback-driven CT. The result of the empirical studies are discussed.

The rest of the paper is organised as follows: Section 2 presents the preliminary background of CT. Section 3 describes our new 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 proposes some further work.

2. BACKGROUND

This section presents some definitions and propositions to give a formal model for CT.

Assume that the Software Under Test (SUT) is influenced by n parameters, and each parameter p_i can take the values from the finite set V_i (i = 1,2,..n). The definitions below are originally defined in [24].

Definition 1. A *test case* of the SUT is a tuple of n values, one for each parameter of the SUT. It is denoted as (v_1, v_2,...,v_n), where $v_1 \in V_1$, $v_2 \in V_2$ ... $v_n \in V_n$.

In practice, these parameters in the test case can represent many factors, such as input variables, run-time options, building options or various combination of them. We need to execute the SUT with these test cases to ensure the correctness of the behaviour of the SUT.

We consider any abnormally executing test case as a *fault.* It can be a thrown exception, compilation error, assertion

failure or constraint violation. When faults are triggered by some test cases, it is desired to figure out the cause of these faults. Some subsets of these test cases 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 other irrelevant parameters are represented as "-".

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

Note that the schema is a formal description of the interaction between parameter values we discussed before.

Definition 3. Let c_l be a l-degree schema, c_m be an m-degree schema in SUT and $l < m$. If all the fixed parameter values in c_l are also in c_m, then c_m *subsumes* c_l. In this case we can also say that c_l is a *sub-schema* of c_m and c_m is a *super-schema* of c_l, which can be denoted as $c_l \prec c_m$.

For example, the 2-degree schema (-, 4, 4, -) is a sub-schema of the 3-degree schema (-, 4, 4, 5), that is, (-,4,4,-) $\prec$ (-,4,4,5).

Definition 4. If all test cases that contain a schema, say c, trigger a particular fault, say F, then we call this schema c the *faulty schema* for F. Additionally, if none of sub-schema of c is the *faulty schema* for F, we then call the schema c the *minimal failure-causing schema (MFS)* [24] for F.

Note that MFS is identical to the failure-inducing interaction we discussed previously. In this paper, the terms *failure-inducing interactions* and *MFS* are used interchangeably. Figuring the MFS out helps to identify the root cause of a failure and thus facilitate the debugging process.

2.1 CT Test Case Generation

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

Many studies in CT focus on how to generate such a test suite with the aim of 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 strategy of the covering array:

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

2) A set of test cases one time: This strategy generates a set of test cases at each iteration. Through mutating the values of some parameters of some test cases in this test set, it focuses 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 in that it does not firstly generate complete test cases [20]. Instead, it first focuses on assigning values to some part of the factors or parameters to cover the schemas that related to these factors, and then fills up the remaining part to form complete test cases.

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

2.2 Identify the failure-inducing interactions

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 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 for the MFS. Hence, more test cases should be generated to identify the MFS. In CT, the main work in fault localization is to identify the failure-inducing interactions. So in this paper we only focus on the MFS identification. Further works of fault localization such as isolating the defect inside the source code will not be discussed.

A typical MFS identification process is shown in Table 1. This example assumes the SUT has 3 parameters, each can take 2 values, and the test case (1, 1, 1) fails. Then in Table 1, as test case t failed, we mutate one factor of the t one time to generate new test cases: t_1 – t_3, It turns out that test case t_1 passed, which indicates that this test case break the MFS in the original test case t. So (1,-,-) should be a 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 example

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 identification process mutate one factor of the original test case at a 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 called the OFOT method, which is a well-known MFS identification method in CT. In this paper, we will focus on this identification method. It should be noted that the following proposed CT framework can be easily applied to other MFS identification methods.

3. MOTIVATING EXAMPLE

In this section, a motivating example is presented to show how traditional CT works as well as its limitations. This example is derived from our attempt to test a real-world software–HSQLDB, which is a pure-java relational database engine with large and complex configuration space. To extract and manipulate valid configurations of this highly-configurable system is important, as different configuration

can result in significant different behaviours of the system [13, 28, 30] (HSQLDB may normally works under some properly configurations, but crashes or throws exceptions under some other configurations, for example).

Considering the large configuration space of HSQLDB, we firstly utilized CT to generate a relatively small set of test cases. Each of them is actually a set of specific assignments to those options we cared. For each configuration, HSQLDB is tested by sending prepared SQL commands. We recorded the output of each run, but unfortunately, about half of them produced exceptions or warnings. Following the schedule of traditional CT, we started the identification process to isolate the failure-inducing option interactions in those failing configurations. Each failing configuration should be individually handled, in principle, as there may exist distinct failure-inducing option interactions among them. However, this successively identification process, although appealing, was hardly ever followed for this case study. This is because there are too many failing configurations and most of them contain the same failure-inducing option interactions, based on which the MFS identification process is wasteful and inefficient.

For the sake of convenience, we provide a highly simplified scenario to illustrate the problems we encountered. Consider four options in HSQLDB – *Server type*, *Scroll Type*, *Parameterised SQL* and *Statement Type*. The possible values each option can take on are shown in Table 2. Based on the report in the bug tracker of HSQLDB [1], an *incompatible exception* will be triggered if a *parameterised sql* is executed as *prepared statement* by HSQLDB. Hence, when option *parameterisedSQL* is set to be *true* and *Statmetent* to be *preparedStatement*, our testing will crash. Besides this failure, there exists another option value which can also crash this database engine. It is when *Scroll Type* is assigned to *sensitive*, as this feature is not supported by this version of HSQLDB [2]. Without these knowledge at prior, we needed to detect and isolate these two failure-inducing option interactions by CT.

Table 2: Highly simplified configuration of HSQLDB

	Option	Values
o_1	Server type	server, web-server, in-process
o_2	Scroll type	sensitive,insensitive, forward-only
o_3	parameterised SQL	true, false
o_4	Statement Type	statement, preparedStatement

Table 3 illustrates the process of traditional CT on this subject. For simplicity of notation, we use consecutive symbols 0, 1, 2 to represent each value of each option (For *parameterisedSQL* and *Statement*, the symbol is up to 1). According to Table 3, traditional CT first generated and executed the 2-way covering array (t_1 – t_9 in the *generation* part). Note that this covering array covered all the 2-degree schemas for the SUT. For example, all the possible values combinations between first and second parameters appear at least one test case.

After detecting that t_1, t_4, and t_7 failed during testing, it is desired to respectively identify the MFS of these failing test cases. For t_1, OFOT method is used to generate four additional test cases (t_{10} – t_{13}), and the MFS (-, 0, - , -) of t_1 is identified (*Scroll Type* is assigned to *sensitive*, respectively). This is because only when changing the second factor of t_1, the extra-generated test case will pass. Then the same process is applied on t_4 and t_7. Finally, we found that the MFS of t_4 is (-, -, -, -), indicating that OFOT failed to determine the MFS (will be discussed later), and the MFS of t_7 is the same as t_1. Totally, for detecting and identifying the MFS in this example, we have generated 12 additional test cases(marked with star).

Table 3: Traditional generation-identification life-cycle

		test case				Outcome
		o_1	o_2	o_3	o_4	
Generation						
	t_1	0	**0**	0	0	Fail
	t_2	0	1	1	1	Pass
	t_3	0	2	1	0	Pass
	t_4	1	**0**	**0**	**1**	Fail
	t_5	1	1	0	0	Pass
	t_6	1	2	1	1	Fail
	t_7	2	**0**	1	1	Pass
	t_8	2	1	0	0	Pass
	t_9	2	2	0	0	Fail
Identification						
t_1 (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
	MFS	**(-, 0, -, -)**				
t_4 (1,0,0,1)	t_{14}*	2	**0**	**0**	**1**	Fail
	t_{15}*	1	1	**0**	**1**	Fail
	t_{16}*	1	**0**	1	1	Fail
	t_{17}*	1	**0**	0	0	Fail
	MFS	**(-, -, -, -)**				
t_7 (2,0,1,1)	t_{18}*	0	0	1	1	Fail
	t_{19}*	2	1	1	1	Pass
	t_{20}*	2	0	0	1	Fail
	t_{21}*	2	0	1	0	Fail
	MFS	**(-, 0, -, -)**				

We refer to such traditional life-cycle as ***Sequential CT*** (SCT). However, we believe this may not be the best choice in practice. The first reason is that the engineers normally do not want 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-cycle can generate many redundant and unnecessary test cases, which negatively impacted on both test cases generation and MFS identification. The most obvious negative effect in this example is that we did not identify the expected failure-inducing interaction (-, -, 0, 1), which corresponds that option *parameterisedSQL* is set to be *true* and *Statmetent* to be *preparedStatement*. More shortcomings of the sequential CT are discussed as following:

3.1 Redundant test cases

[1]details see: http://sourceforge.net/p/hsqldb/bugs/1173/
[2]details see: http://hsqldb.org/doc/guide/guide.html

The first shortcoming of SCT is that it may generate redundant test cases, such that some of them do not cover as more uncovered schemas as possible. As a consequent, SCT may generate more test cases than actually needed. This can be reflected in the following two aspects:

1) The test cases generated in the identification stage can also contribute some coverage, i.e., the schemas appear in the passing test cases in the identification stage may have already been covered in the test cases generation stage. For example, when we identify the MFS of t_1 in Table 3, 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 other word, if we firstly identify the MFS of t_1, then t_2 is not a good choice as it does not covered as many 2-degree schemas as possible. For example, (1, 1, 1, 1) is better than this test case at contributing more coverage.

2)The identified MFS should not appear in the following generated test cases. This is because according to the definition of MFS, each test case containing 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 detection. Take the example in Table 3, after identifying the MFS – (-, 0, -, -) of t_1, we should not generate the test case t_4 and t_7. This is because they also contain the identified MFS (-, 0, -, -), which will result in them failing as expected. Since the expected failure caused by MFS (-, 0, -, -) makes t_7 and t_9 superfluous for error-detection, the additional test cases (t_{14} to t_{21}) generated for identifying the MFS in t_4 and t_7 are also not necessary.

3.2 Multiple MFS in one test case

Multiple MFS will negatively affect the accurateness of MFS identification – and even make it hardly obtain a validate schema. For example, there are two MFS in t_4 in Table 3, i.e., (-, 0, -, -) and (-, -, 0, 1) (in bold). When we use OFOT method, we found all the extra-generated test cases (t_{14} to t_{17}) failed. These outcomes give OFOT a false indication that all the failure-inducing factors are not broken by mutating those four parameter values. As a result, OFOT cannot determine which schemas are MFS, which is denoted as (-, -, -, -).

The reason why OFOT cannot properly work is that this approach can only break one MFS at a time. If there are multiple MFS in one test case, the extra-generated test cases will always fail as they contain other non-broken MFS (see bold parts of t_{14} to t_{17}). Some approaches are proposed to handle this problem, but either can not handle multiple MFS that have overlapping parts [35], or consume too many extra-generated test cases [31, 27]. So in practice, to make MFS identification more effective and efficient, we need to avoid the appearance of multiple MFS in one test case.

SCT, however, does not offer much support for this concern. This is mainly because it is essentially a post-analysis framework, i.e., the analysis for MFS comes after the completion of test cases generation and execution. As a result, in the generation stage, testers have no knowledge of the possible MFS, and surely it has opportunities that multiple MFS appear in one test case.

3.3 Masking effects

Traditional covering array usually offer an inadequate testing due to *Masking effects* [10, 33]. A masking effect[33] is an effect that some failures or exceptions prevent a test case from testing all valid schemas in that test case, which the test case is normally expected to test. For example in Table 3, t_1 is initially expected to cover all the 2-degree schemas –(0, 0, - , -), (0, -, 0, -), (0, -, -, 0), (-, 0, 0, -), (-, 0, -, 0), (-, -, 0, 0), but when we found that it failed during testing, this conclusion is no longer that sure. This is because, the failing of t_1 (*ScrollType* is set to be *sensitive*) crashed HSQLDB, and as a result, it did not go on executing the remaining test code, which may affect the examination of some interactions of t_1. Hence, we cannot ensure we thoroughly exercise all the interactions in this failing test case.

Since traditional covering array cannot reach a adequate testing, as an alternative, *tested t-way interaction criterion* as a more rigorous coverage standard is proposed [33]. This criterion, to be simply, shows that a t-degree schema is covered iff (1) it appears in a passing test case (2)it is identified as MFS or faulty schema.

Now it is surely that this criteria is not satisfied with traditional covering array alone, next let us examine whether this criterion can be satisfied with SCT, i.e., too see if *tested t-way interaction criterion* can be satisfied with the combination of traditional covering array and MFS identification. One obvious insight is that if there is only single MFS in each failing test case, this criterion is satisfied. For example in Table 3, t_1 contained a single MFS (-, 0, -, -), and we identified this MFS by generating four extra test cases (t_{10} to t_{13}). As for t_1, the schemas (-, 0, -, -) can be deemed to be covered.

Hence, to get an adequate testing, we should not only generate test cases. This coverage, however, is called t- coverage.

This problem can be more. Take example , the covering arry t_1 to t_9. So

But with MFS identification, this problem can be relived. For single MFS, however, this can.

But for multiple MFS, however, this problem is worse.

This is mainly caused after the second.

Even worse, such test case may suffer from the *masking effects* [33], as failures caused by the already identified MFS can prevent the test case from normally checked (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.

and even worse 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.

4. INTERLEAVING APPROACH

To overcome such deficiencies in traditional CT, we propose a new CT generation-identification framework – ***Interleaving CT***. Our new framework aims at enhancing the interaction of generation and identification to reduce the unnecessary and invalid test cases discussed previously.

4.1 Overall framework

The basic outline of our framework is illustrated in Figure 2. Specifically, this new framework works as follows: First, it checks whether all the needed schemas are 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 current coverage is not satisfied, it will generate a new

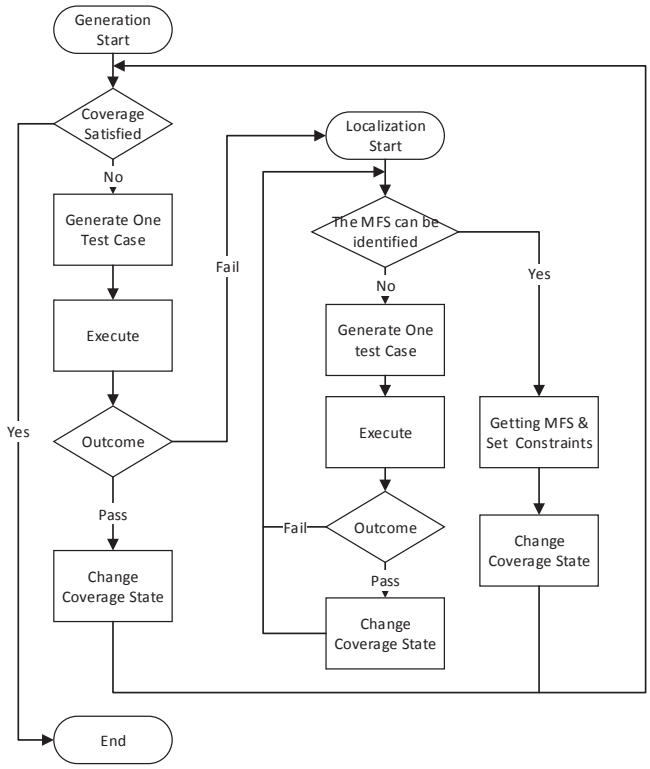

Figure 2: The Interleaving Framework

test case to cover as many uncovered schemas as possible. After that, it will execute this test case with the outcome of the execution either be pass (executed normally, i.e., does not triggered an exception, violate the expected oracle or the like) or fail (on the contrary). When the test case passes 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 was not 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 identification module to identify 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, as we can not figure out which schemas are responsible for this failure among all the schemas in this test case until we identify them.

The identification module works in a similar 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 difference from traditional MFS identifying process is that we record the coverage that this module has 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 has 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 (Otherwise the test case must fail and it cannot contribute to more coverage), and second all the t-degree schemas that are *related* to these MFS as covered. Here the *related* schemas indicate the following three types of t-degree schemas:

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

Second, the schemas that are the **super-schemas** of these MFS. By definition of the super-schemas (Definition 3), if the test case contains the super-schemas, it must also contain all its sub-schemas. So every test case that contains the super-schemas of the MFS must fail after execution. As a result, they will never be covered as we do not change the 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 containing this schema will fail and so we will not change the coverage state. In fact, by Definition 4, they can be deemed as faulty schemas.

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 can set them as covered after the execution of the identification module, so that the overall process can stop.

4.2 Modifications of CT activities

More details of the modifications of CT activities are as follows:

(1) *Modified CT Generation* : We adopt the *one test case one time* method as the basic skeleton of the generation process. Originally, the generation of one test case can be formulated as EQ1.

$$t \leftarrow select(\mathcal{T}_{all}, \Omega, \xi) \quad \text{(EQ1)}$$

There are three factors that determine the selection of test case t. $\mathcal{T}_{all}$ represents all the valid test cases that can be selected to be executed. Usually the test cases that have been tested will not be included as they have no more contribution to the coverage. Ω indicates the set of schemas that have not been covered yet. ξ is a random factor. Most CT generation approaches prefer to select a test case that can cover as many uncovered schemas as possible. This greedy selection process does not guarantee an optimal solution, i.e., the final size of the set of test cases is not guaranteed to be minimal. The random factor ξ is used to help to escape from local optimum.

As discussed in section 3, we should make the MFS not appear in the test cases generated afterwards, by treating them as the forbidden schemas. In other words, the candidate test cases that can be selected are reduced, because those test cases that contain the currently identified MFS should not appear next. Formally, let $\mathcal{T}_{MFS}$ indicates the set of test cases that contain the currently identified MFS, then the test case selection is augmented as EQ2.

$$t \leftarrow select(\mathcal{T}_{all} - \mathcal{T}_{MFS}, \Omega, \xi) \quad \text{(EQ2)}$$

In this formula, the only difference from EQ1 is that the

candidate test cases that can be selected are changed to $\mathcal{T}_{all} - \mathcal{T}_{MFS}$, which excludes $\mathcal{T}_{MFS}$ from candidate test cases.

In practice, it is impossible to thoroughly search the exhaustive candidate test cases $\mathcal{T}_{all}$ to get a specific test case t. Hence, some heuristic methods are used to simplify the selection. For example, AETG [3] successively assigns the value to each parameter to form a test case (in random order). The value to be assigned is the one that appears in the greatest number of uncovered schemas. Correspondingly, to exclude the test cases in $\mathcal{T}_{MFS}$, it is common to utilize a constraint solver to avoid the forbidden schemas [6, 7].

(2) *Modified identification of MFS* : Traditional MFS identification aims at finding the MFS in a failing test case. As discussed before, test cases in the covering array are not enough to identify the MFS. Hence, additional test cases should be generated and executed in this CT activity. Generally, an additional test case is generated based on the original failing test case, so that the failure-inducing parts can be determined by comparing the difference between the additional test cases and the original failing test cases. Take the OFOT approach as an example. In Table 3, the additional test case t_{11} is constructed by mutating the second parameter value of the original failing test case t_1. Then as t_{11} passed the testing, we can determine that the second parameter value (-, 0, -, -) must be a failure-inducing element. Formally, let $t_{failing}$ be the original failing test case, Δ be the mutation parts, $\mathcal{P}$ be the parameters and their values, then the traditional additional test case generation can be formulated as EQ3.

$$t \leftarrow mutate(\mathcal{P}, t_{failing}, \Delta) \quad \text{(EQ3)}$$

EQ3 indicates that the test case t is generated by mutating the part Δ of the original failing test case $t_{failing}$. Note that the mutated values may have many choices, as long as they are within the scope of $\mathcal{P}$ and different from those in $t_{failing}$. For example, for the original failing test case t_1 (0,0,0,0) in Table 3, let Δ be the second parameter value, then test cases (0, 1, 0, 0) and (0, 2, 0, 0) all satisfy EQ3. We refer to all the test cases that satisfy EQ3 as $\mathcal{T}_{candidate}$, which can be formulated as EQ4.

$$\mathcal{T}_{candidate} = \{\ t \mid t \leftarrow mutate(\mathcal{P}, t_{failing}, \Delta)\ \} \quad \text{(EQ4)}$$

Traditional MFS identification process just selects one test case from $\mathcal{T}_{candidate}$ randomly. However, to adapt the MFS identification process to the new CT framework, this selection should be refined.

Specifically, there are two points to note. First, the additional test case should not contain the currently identified MFS; second, the additional test case is expected to cover as many uncovered schemas as possible. These two goals are similar to CT generation, hence we can directly apply the same selection method on additional test case generation, which can be formulated as EQ5. The same as EQ2, EQ5 excludes the test cases that contain the currently identified MFS from the candidate test cases ($\mathcal{T}_{candidate} - \mathcal{T}_{MFS}$), and selects the additional test case which covers the greatest number of uncovered schemas (Ω).

$$t \leftarrow select(\mathcal{T}_{candidate} - \mathcal{T}_{MFS}, \Omega, \xi) \quad \text{(EQ5)}$$

(3) *Updating uncovered schemas* : After the MFS are identified, some related t-degree schemas, i.e., *MFS themselves*, *super-schemas* and *implicit forbidden schemas*, should be set as covered to enable the termination of the overall CT process. The algorithm that seeks to handle these three types of schemas is listed in Algorithm 1.

Algorithm 1 Changing coverage after identification of MFS

Input: $\mathcal{S}_{MFS}$ ▷ currently identified MFS
Ω ▷ the schemas that are still uncovered
$\mathcal{T}_{all}$ ▷ all the possible test cases
$\mathcal{T}_{MFS}$ ▷ all the test cases that contain the MFS
Output: *void* ▷ do not need any output

1: **for each** $s \in \mathcal{S}_{MFS}$ **do**
2: **if** s is t-degree schema **then**
3: $\Omega \leftarrow \Omega \backslash s$
4: **end if**
5: **for each** s_p is super-schema of s **do**
6: **if** s_p is t-degree schema **then**
7: $\Omega \leftarrow \Omega \backslash s_p$
8: **end if**
9: **end for**
10: **end for**
11: **for each** $s \in \Omega$ **do**
12: **if** $\nexists t \in (\mathcal{T}_{all} - \mathcal{T}_{MFS}), s.t., t.contain(s)$ **then**
13: $\Omega \leftarrow \Omega \backslash s$
14: **end if**
15: **end for**

In this algorithm, we firstly check each MFS (line 1) to see if it is t-degree schema (line 2). We will set those t-degree MFS as covered and remove them from the uncovered schemas set Ω (line 3). This is the first type of schemas –*themselves*. For each t-degree super-schema of these MFS, it will also be removed from the uncovered schemas set (line 5 - 9), as they are the second type of schemas – *super-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 11), and check if it is the implicit forbidden schema (line 12). The checking process involves solving a satisfiability problem. Specifically, if we can not find a test case from the set ($\mathcal{T}_{all} - \mathcal{T}_{MFS}$) (excluding those that contain MFS), such that it contains the schema under checking, then we can determine the schema is the implicit forbidden schema and it needs to be removed from uncovered schemas set (line 13). This is because in this case, the schema under checking can only appear in $\mathcal{T}_{MFS}$, which we will definitely not generate in later iterations. In this paper, a SAT solver will be utilized to do this checking process.

4.3 Problems that solved

(1) redundant test cases
as we have compute the coverage.
(2) As we label them as forbidden schemas
(3) Masking effects

4.4 Example

With this new framework, when we re-consider the example in Table 3 in section 3, we can get the result listed in Table 4.

This table consists of two main columns, in which the left indicates the generation part while the right column indicates the identification process. We can find that, after identifying the MFS (-, 0, -, -) for t_1, the following test cases (t_6 to t_{13}) will not contain this schema. Correspondingly, all

Table 4: Interleaving CT case study

Generation						*Identification*					
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						

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 identification process cover six *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, when using the interleaving CT approach, the overall generated test case are 8 less than that of the traditional sequential CT approach in Table 3.

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

5. LIMITATIONS OF OUR APPROACH

Alougth our approach provides a more adaptive and flexible framework for CT, there are several issues that needs to be concerned.

5.1 Incorrectly identified schemas

incorrect schemas can give us unrelieble results. However, this is inavoid. To definely get exact MFS, according to charle courblun and Nie , we need to exhaustive conducnt every possible test cases, which is however, not possible. So in practice, we get an ¡ [Zeller's paper to show how global minimal is not possible].

Based on this, if our algorithm get an wrong MFS, then it can be .

Possible solution 1) Multiple approaches combined, this will enhance the şř of our method. As we can learn in the emprically study, the combination of FDA-CI can significantly improve the prisce of the MFS identification and testing adquattely.

2) More critically feedback validation. That is, when a MFS is identified. We need to send to the developers to see whether it is one of the root cause of failure. And if correct, then we go on. Else, we will re-compute the MFS with the developers information.

5.2 Constraints handling

As to avoid the idenified MFS is needed. In this paper, we handle this as constraints, which needs to get an solver to aviod the following result. This handling, however, obviously increase the cost of the whole CT process. As constraints satification (SAT) problem and optimcal problem (covering array) is both tough [][][][][], the increasing is possible.

Possible solution

1) Illustrate history to reducation

2) better practical stop [Zeller' paper saied a good term, to show why isolation)

3) better constratins modeling

6. EMPIRICALLY STUDY

6.1 software subjects

6.2 compare with traditional test cases

test cases

redundant test cases (generate and identify)

6.3 multiple MFS

multiple failures in one test case

f-measure

overlap and non overallp

6.4 masking effects

masking effects condition.

integrated. using FDA-CIT to make it adequte testing

7. EMPIRICAL STUDIES

To evaluate the effectiveness and efficiency of the interleaving CT approach, we conducted a series of empirical studies on several open-source software subjects.

7.1 Subject programs

The five subject programs used in our experiments are listed in Table 5. Column "Subjects" indicates the specific software. Column "Version" indicates the specific version that is used in the following experiments. Column "LOC" shows the number of source code lines for each software. Column "Faults" presents the fault ID, which is 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 (For subjects written in more than one programming languages, only the main programming language is shown).

Table 5: 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	v_1	173	#Seed	c

Among these subjects, Tomcat is a web server for java servlet; Hsqldb is a pure-java relational database engine; Gcc is a programming language compiler; Jflex is a lexical analyzer generator; and Tcas is a module of an aircraft collision avoidance system. We select these software as subjects 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 can reach 460800 [29, 16].

As the target of our empirical studies is to compare the ability of fault defection between our approach with traditional ones, we firstly must know these faults and their corresponding MFS in prior, so that we can determine whether the schemas identified by those approaches are accurate or not. For this, we looked through the bug tracker of each software and focused on the bugs which are caused by the interaction of configuration options. Then for each such bug, we 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 used as a experimental object for the fault detection studies.

7.1.1 Specific inputs models

To apply CT on the selected 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 experimental time and computing resource. Instead, a moderate small set of these configuration options is selected. It includes the options that cause the specific faults in Table 5, thus the test cases generated by CT can detect these faults. Additional options are also included to create some noise for the MFS identification approach. These options are selected randomly. Details of the specific options and their corresponding values of each software are posted at `http://barbie.uta.edu/data.htm`. A brief overview of the inputs models as well as the corresponding MFS (degree) is shown in Table 6.

Table 6: 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 number of MFS (in the parentheses) with that corresponding degree.

7.2 Comparing with sequential CT

After preparing the subjects software, next we constructed the experiment to evaluate the efficiency and effectiveness of our approach. To this aim, we need to compare our framework with the traditional sequential CT approach to see if interleaving CT approach has any advantage.

The covering array generating algorithm used in the experiment is AETG [3], as it is the most common one-test-case-one-time generation algorithm. And the MFS identifying algorithm is OFOT [24] as discussed before. The constraints handling solver is a java SAT solver – SAT4j [19].

7.2.1 Study setup

For each software except *Tcas*, a test case is determined to be passing if it ran without any exceptions; otherwise it is regarded as failing. For *Tcas*, as the fault is injected, we determine the result of a test case by separately running it with the original correct version and the mutation version. The test case will be labeled as passing if their results are the same; otherwise, it is deemed as failing.

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 then evaluate the results. In other word, for each subject software, we will repeatedly execute traditional approach and our approach 30 times to detect and identify the MFS.

To evaluate the results of the two approaches, one metric is the cost, i.e., the number of test cases that each approach needs. Specifically, the test cases that are generated in the CT generation and MFS identification, respectively, are recorded and compared for these two approaches. Apart from this, another important metric is the quality of their identified MFS. For this, we used standard metrics: *precision* and *recall*, which are defined as follows:

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

and

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

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

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

7.2.2 Result and discussion

Table 7 presents the results for the number of test cases. In Column 'Method', *ict* indicates the interleaving CT approach and *sct* indicates the sequential CT approach. The results of three covering criteria, i.e., 2-way, 3-way, and 4-way are shown in three main columns. In each of them, the number of test cases that are generated in *CT generation* activity (Column 'Gen'), in *MFS identification* activity (Column 'Iden'), and the total number of test cases (Column 'Total) are listed.

One observation from this table is that the total number of 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 of test cases for 3-way and 4-way coverage. For *Tcas*, however, as the MFS are hard to detect (all of them have degrees greater than 9), so both approaches nearly do not trigger errors. Under this condition, both approaches will be transferred to a normal covering array.

The gap of total number of test cases is mainly due to the difference of the number of test cases generated in *MF-*

S identification activity. In fact, their results in the *CT generation* are almost the same. Even for $4-way$ coverage criteria which may needs thousands test cases, the gap between them are no more than 10.

For *MFS identification* activity, the interleaving approach only consumed a relatively small amount of test cases when comparing to the sequential approach. What's more, the interleaving approach obtained almost the same results under the 2-way, 3-way, and 4-way coverage (see the cells in bold). This is as expected, as the MFS identification only happens after a test case fails in our interleaving approach. And after the identification process, the identified MFS will be set as forbidden schemas for the latter generated test cases. As a result, each MFS only needs to be identified once, no matter what the coverage is and how many test cases needed to be generated.

However, this is not the case for the sequential approach. As discussed before, the sequential approach does not identify the MFS at early iteration, so that these MFS can appear in latter test cases. As a result, it needs many more test cases to identify the same MFS , which is a huge waste. Even worse, the more test cases generated in *CT generation* activity, the more test cases are needed in *MFS identification* activity (See the Column 'Iden' of sequential approach under 2-way, 3-way and 4-way coverage). This is because the possibility that failures are triggered is increased when there are more test cases without forbidding the appearance of these MFS.

With regarding to the quality of the identified MFS, the comparison of the two approaches are listed in Table 8. Based on this table, 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 is 0.27 (2-way for Tomcat), and the average gap is 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 contains multiple MFS. Our previous study showed that multiple MFS in a test case can reduce the accuracy of the MFS identifying algorithms [27]. 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, they will significantly impact the following iteration. This is because we will compute the coverage and forbidden schemas based on previous identified MFS. It was the other way around for the traditional approach. It suffers when a test case contains multiple MFS, but correspondingly, previous identified MFS has little influence on the traditional approach.

In summary, our approach needs much less test cases than traditional sequential CT approach, and there is no decline in the quality of the identified MFS when comparing with traditional approach.

7.3 Comparing with Error Locating Array

Error locating array[9, 23] is a well-designed set of test cases that can support not only failure detection, but also the identification of the MFS of the failure. It is known that only with a covering array sometimes is not sufficient to identify the MFS, thus additional test cases are needed. Martínez et al.[22] have proved that a $(t+d)$-way covering array can identify all the MFS with the number of them no more than d, and degree 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. So with the number d and degree t known in prior, a $(t+d)$-way covering array is an *Error Locating Array (ELA)*.

To compare our approach with this Error Locating Array is meaningful, as both approaches have the same target. The relationship between our approach with the Error Locating Array can be deemed as the dynamic vs static. In detail, our approach dynamically detects and identifies 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 generates a static covering array, and it can support MFS identification if the number and degree of these schemas are known in prior.

7.3.1 Study setup

As the results of our approach have already been collected as described in Section 5.2, this section will just apply ELA to identify the MFS of the 5 subjects in Table 5. 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 in any part of these MFS. In our experiment, all the five subject programs satisfy this condition. Based on this, we then applied ELA to generate appropriate covering arrays for each subject program and recorded the MFS identified as well as the overall test cases generated. The covering array generation algorithm we adopted in this experiment is also AETG [3], and similar as previous experiments, this experiment is repeated 30 times.

7.3.2 Result and discussion

The number of overall test cases and the quality of the identified MFS are listed in Table 9. We can firstly observe that this approach needs more test cases than the two approaches discussed before. This is as expected, as this approach needs to generate a higher-way covering array than the previous two approaches. Apart from the high cost, this approach correctly identified all the real MFS. The accuracy has been proved in [22, 23]. Note that this perfect MFS identification result is based on the fact that it knows the number and degree of the MFS, which is usually not available in practice.

Table 9: Results from Error Locating Array

Subject	Size	Precision	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*) by dividing *f-measure* by *size*. The normalized result of this combination metric is shown in Fig.3. In this figure, *ict* and *sct* represent interleaving CT approach and traditional sequential CT approach, respectively. *Ela* indicates the error locating array approach.

Table 7: Comparison of the number of test cases

Subjects	Method	2-way			3-way			4-way		
		Gen	Iden	Total	Gen	Iden	Total	Gen	Iden	Total
Tomcat	ict	12.73	**30**	42.73	34.97	**29.67**	64.64	85.57	**29.33**	114.9
	sct	14.1	105.6	119.7	38.3	256.03	294.33	93.4	578.4	671.8
Hsqldb	ict	14.87	**12**	26.87	46.27	**18.4**	64.67	128.87	**14.4**	143.27
	sct	15.83	10.8	26.63	47.8	33.6	81.4	130.07	103.23	233.3
Gcc	ict	14.5	**19.33**	33.83	47.9	**21.33**	69.23	102.63	**24.67**	127.3
	sct	15.27	19.67	34.94	50.97	64.93	115.9	103.87	124.6	228.47
Jflex	ict	15.8	**14**	29.8	50.47	**14**	64.47	145.17	**14**	159.17
	sct	15.83	28.93	44.76	49.63	134.74	184.37	142.1	433.47	575.57
Tcas	ict	109.03	**0**	109.03	426.8	**0.8**	427.6	1629.83	**3.2**	1633.03
	sct	108.67	0	108.67	426.77	1.2	427.97	1633.2	2.8	1636

Table 8: Comparison of the quality of the identified MFS

Subjects	Method	2-way			3-way			4-way		
		Precision	Recall	F-measure	Precision	Recall	F-measure	Precision	Recall	F-measure
Tomcat	ict	1	1	**1**	0.97	0.96	**0.96**	0.93	0.91	**0.92**
	sct	0.73	0.74	0.73	0.75	1	0.86	0.75	1	0.86
Hsqldb	ict	0.67	0.4	**0.49**	0.37	0.37	0.37	0.37	0.37	0.37
	sct	0.55	0.3	0.38	0.5	0.5	0.5	0.5	0.5	0.5
Gcc	ict	0.47	0.28	**0.34**	0.46	0.37	**0.40**	0.58	0.48	**0.52**
	sct	0.33	0.18	0.23	0.36	0.43	0.39	0.33	0.5	0.40
Jflex	ict	1	1	1	1	1	1	1	1	1
	sct	1	1	1	1	1	1	1	1	1
Tcas	ict	0	0	0	0.03	0.0007	**0.001**	0.07	0.001	**0.0027**
	sct	0	0	0	0	0	0	0.03	0.001	0.0026

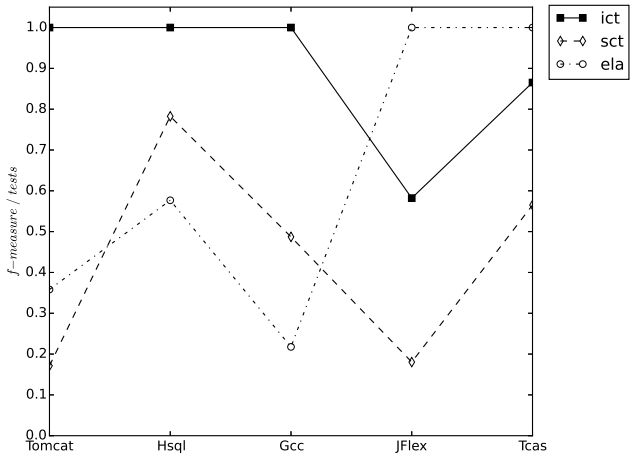

Figure 3: Comparison of the combining metric

We can learn that *ict* performs better than *sct* for all five subjects. For *ela*, our approach performs better for three subjects *Tomcat*, *Hsqldb*, and *Gcc*. The reason that our approach does not perform as well as *ela* for subject *JFlex* is that the MFS for that object is a single 2-degree schema (see Table 6), under which ela just needs a 3-way covering array. For subject *Tcas*, with high-degree MFS as discussed before, our approach is hard to trigger an error with only *2-way*, *3-way*, and *4-way* covering array. As a result, our approach can hardly identify the MFS. Apart from these two special cases, our approach has a significant advantage over the ELA approach.

To summarize, ELA gets the best quality of the MFS, but needs much more test cases than our approach. It also needs to know the number and degree of the MFS in prior, which limits its application in practice.

7.4 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 plan to conduct 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, there have been many more generation algorithms and MFS identification algorithms. In our empirical studies, we just used AETG [3] as the test case generation strategy, and OFOT [24] as the MFS identification strategy. As different generation and identification algorithms may affect the performance our proposed CT framework, especially on

the number of test cases, some studies for different test case generation and MFS identification approaches are desired.

8. RELATED WORKS

Combinatorial testing has been widely applied in practice [15], especially on domains like configuration testing [32, 8, 11] and software inputs testing [3, 1, 12]. A recent survey [25] 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 MFS identification are two most important parts in CT studies.

Although CT has been proven to be effective at detecting and identifying the interaction failures in SUT, however, to directly apply them in practice can be inefficient and some times even does not work at all. Some problems, e.g., constraints of parameters values in SUT [5, 7], masking effects of multiple failures[10, 33], dynamic requirement for the strength of covering array [11], will bring many troubles to CT process. To overcome these problems, some works try to make CT more adaptive and flexible.

JieLi [21] augmented the MFS identifying algorithm by selecting one previous passing test case for comparison, such that it can reduce some extra test cases when compared to another efficient MFS identifying algorithm [35].

S.Fouché et al., [11] introduced the notion of incremental covering. Different from traditional covering array, it does not need a fixed strength to guide the generation; instead, it can dynamically generate high-way covering array based on existing low-way 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-degree 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 the masking effects between multiple failures. Yilmaz [33] extended that work by refining the MFS diagnosing method. Specifically, this feedback driven approach firstly generate a t-way covering array, and after executing them, the MFS will be identified by utilizing a classification tree method. It then forbidden these MFS and generate additional test cases to cover the interactions that are masked by the MFS. This process continues until all the interactions are covered. Additionally, Nie [26] constructed an adaptive combinatorial testing framework, which can dynamically adjust the inputs model, strength t of covering array, and generation strategy during CT process.

Our work differs from them mainly in that we proposed a highly interactive framework for test case generation and MFS identification. Specifically, we do not generate a complete t-way covering array at first; instead, when a failure is triggered by a test case, we immediately terminate test cases generation and turn to MFS identification. After the MFS is identified, the coverage will be updated and the test cases generation process continues.

9. CONCLUSIONS

Combinatorial testing is an effective testing technique at detecting and diagnosis the failure-inducing interactions in the SUT. Traditional CT separately studies test cases generation and MFS identification. In this paper, we proposed a new CT framework, i.e., *interleaving CT*, that integrates these two important stages, which allows for both generation and identification better share each other's information. As a result, interleaving CT approach can provide a more efficient testing than traditional sequential CT.

Empirical studies were conducted on five open-source software subjects. The results show that with our new CT framework, there is a significant reduction on the number of generated test cases when compared to the traditional sequential CT approach, while there is no decline in the quality of the identified MFS. Further, when comparing with the ELA [23, 22], our approach also performed better, especially with fewer test cases.

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

10. 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] D. Jin, X. Qu, M. B. Cohen, and B. Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 215–224. ACM, 2014.

[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. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, pages 91–95. IEEE, 2002.

[18] D. R. Kuhn, D. R. Wallace, and J. AM Gallo. Software fault interactions and implications for software testing. *Software Engineering, IEEE Transactions on*, 30(6):418–421, 2004.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] C. Song, A. Porter, and J. S. Foster. itree: Efficiently discovering high-coverage configurations using interaction trees. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 903–913. IEEE Press, 2012.

[31] Z. Wang, B. Xu, L. Chen, and L. Xu. Adaptive interaction fault location based on combinatorial testing. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 495–502. IEEE, 2010.

[32] 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.

[33] 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.

[34] 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.

[35] 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.