Identifying minimal failure-inducing schemas for multiple faults*

[Extended Abstract] †

Xintao Niu
State Key Laboratory for Novel Software Technology
Nanjing University
China, 210093
niuxintao@smail.nju.edu.cn

Changhai Nie
State Key Laboratory for Novel Software Technology
Nanjing University
China, 210093
changhainie@nju.edu.cn

ABSTRACT

Combinatorial testing(CT) is proven to be effective to reveal the potential failures caused by the interaction of the inputs or options of the system under test(SUT). A key problem in CT is to isolate the failure-inducing interactions in SUT as it can facilitate the debugging effort by reducing the scope of code that needs to be inspected. Many algorithms has been proposed to identify the failure-inducing interactions in SUT, however, most of these studies either just consider the condition of one fault in SUT or ignore masking effects among multiple faults which can bias their identified results. In this paper, we analysed how the masking effect of multiple faults affect on the isolation of failure-inducing interactions. We further give a strategy of selecting test cases to alleviate this impact. The key to the strategy is to prune these test cases that may trigger masking effect and generate no-masking-effect ones to test the interactions supposed to be tested in these pruned test cases. The test-case selecting process repeated until we get enough information to isolate the failure-inducing interactions in SUT. We conducted some empirical studies on several open-source GNU software. The result of the studies shows that multiple faults do exist in real software and our approach can assist combinatorial-based failure-inducing identifying methods to get a better result when handling multiple faults in SUT.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

Minimal failure-inducing schemas, Masking effect

*(Does NOT produce the permission block, copyright information nor page numbering). For use with ACM_PROC_ARTICLE-SP.CLS. Supported by ACM.

†A full version of this paper is available as *Author's Guide to Preparing ACM SIG Proceedings Using LaTeX2$_\epsilon$ and BibTeX* at www.acm.org/eaddress.htm

1. INTRODUCTION

With the increasing complexity and size of modern software, many factors, such as input parameters and configuration options, can influence the behaviour of the system under test(SUT). The unexpected faults caused by the interaction among these factors can make testing such software a big challenge if the interaction space is too large. Combinatorial testing can systematically sample the interaction space and select a relatively small size of test cases that cover all the valid iterations with the number of factors involved in the interaction no more than a prior fixed integer, i.e., strength t.

Once failures are detected, it is desired to isolate the failure-inducing interactions in these failing test cases. This task is important in CT as it can facilitate the debugging effort by reducing the code scope that needed to inspected. Many algorithms has been proposed to identify the failure-inducing interactions in SUT, which include approaches such as building classification tree model [6], generating one test case one time [5], ranking suspicious interactions based on prior rules[3], using graphic-based deduction [4] and so on. These approaches can be partitioned into two categories [1]: *adaptive*–tests cases are chosen based on the outcomes of the executions of prior tests or *nonadaptive*–test cases are chosen independent and can be executed parallel.

Recent work [2] presented the masking effects . Additionally, [7] illustrated the impact on the failure-inducing identifying approaches CTA. In this paper, we first formulated the impact come from the masking effects, based on this we proposed a strategy to . We have

When applied these approaches on several open-source software, we found that These approaches have their tedian and can good at some changjing or assumpitions work well. In our recent studies on some open-source software, however, we found they didn't behave as expected when . Thorough a in-depth analysis, we learned the reason why they didn't behave as expected is that most of these approaches mainly consider the SUT just contain one fault, i.e., the oracle of the test case either be false or pass. Put aside the performance these approaches exhibited for single fault, however, in our recently studies on several GNU open-source software, we find these methods didn't behave as expected when these software contain multiple faults. The main reason why these methods fails to behave normally is that they didn't consider the masking effect that may happens among different faults. Take the Linux command– Grep for example, we noticed that there are two different faults reported in the bug tracker system. The first one claim that Grep incorrectly match unicode patterns with '\<\>', while the second one claim a incompatibility between option '-c' and '-o'. When we put this two scenario into one test case only one fault information will be observed, which means another fault is masked by the observed one. Obviously this fact do trouble these algorithms as it make them unable to judge whether the test case under testing trigger only the fault observed or triggered both two faults. As a result, it will make wrong analysis in the failure-inducing interactions for these masked faults.

One insight in this paper is that we cannot completely get away from the impact of the masking effect even if we do exhaustive testing. Furthermore, both ignoring the masking effects and regarding multiple faults as one fault is harmful for failure-inducing combinations identifying process. Based on the insight we proposed a strategy to. One remedy to alleviate this problem is to select test cases to reduce the bias, i.e., select test cases that without suffering the masking effect to get a partial but masking-avoiding result. However, most selecting strategy in CT is to cover the interactions with the number of test cases as small as possible. So in this paper we proposed a strategy for selecting test cases with the aim to alleviate the masking effect.

The key to the strategy is to prune test cases that may trigger a masking effect and then select or generate other test cases to test the interactions which were supposed to be tested in these pruned ones. We will keep these pruned test cases for the next iteration to isolate the failure-inducing interactions in these test cases. The process repeated until we characterize all the interactions for each fault. A point need to be noted is that these interactions supposed to be tested in the pruned test cases for one interaction vary in different algorithms we adopted. For example, for the classified tree method, we will generate more test cases that keep the same coverage, and for the one fact one time, we will generate test cases to keep the same.

We surveyed several open-source software by studying the bug reports in their bug tracker system. We further built testing model to reproduce some bugs posted in the tracker system. We augment and find that multiple faults do exist in software of the same version. To evaluate the effectiveness of our approach, we take software benchmark from SIR with the ability to inject bugs into the source code. Based on this controllable subject we choose three different failure-inducing interactions identifying algorithms to compare the performance between the original algorithm and its variation extended with our strategy. The results shows that the algorithm extend with our test cases selecting strategy can perform better than before.

The main contributions of this paper are:

1. We studied the impact on the isolation of the failure-inducing interactions when the SUT contain multiple faults which can mask each other.
2. We proposed a strategy of selecting test cases to reduce the impact of masking effect.
3. We empirically studies our strategy and find that our approach can get a better result.

The rest of this paper is organised as follows: section 2 gives a simple example to motivate our work. Section 3 give some background of the work. Section 4 describe our approach in detail. Section 5 illustrate the experiment and reports the result. Section 6 discusses the related works. Section 7 provides some concluding remarks.

2. MOTIVATION EXAMPLE

Following we have construct a example to illustrate the motivation of our approach. Assume we have a method *foo* which has four input parameters : *a, b, c, d*. The types of these four parameters are all integers and the values that they can take are: $d_a = \{7, 11\}, d_b = \{2, 4, 5\}, d_c = \{4, 6\}, d_d = \{3, 5\}$ respectively. The detail code of this method is listed as following:

```
public static float foo(int a, int b, int c, int d){
  //step 1 will cause a exception when b == c
  float x = (float)a / (b - c);

  //step 2 will cause a exception when c < d
  float y = Math.sqrt(c - d);

  return x+y;

  }
```

Inspecting the simple code above, we can find two faults: First, in the step 1 we can get a ArithmeticException when b is equal to c, i.e., b = 4 & c = 4, that makes division by zero. Second, another ArithmeticException will be triggered in step 2 when c < d, i.e., c = 4 & d = 5, which makes square roots of negative numbers. So the expected MFSs in this example should be (-, 4, 4, -) and (-, -, 4, 5).

Traditional MFS identifying algorithms do not consider the detail of the code. They take black-box testing of this program, i.e., feed inputs to those programs and execute them to observe the result. The basic justification behind those approaches is that the failure-inducing schema for a particular fault must only appear in those inputs that trigger this fault. As traditional MFS identifying algorithms aim at using as small number of inputs as possible to get the same or approximate result as exhaustive testing, so the results derive from a exhaustive testing set must be the best that these MFS identifying approaches can reach. Next we will illustrate how exhaustive testing works on identifying the MFS in the program.

We first generate every possible inputs as listed in the Column "test inputs" of table 1, and execute them to get the result listed in Column "result" of table 1. In this Column, "PASS" means that the program runs without any exception under the inputs in the same row. "Ex 1" indicate that the program encounter a exception corresponding to the step 1 and "Ex 2" indicate the program trigger a exception corresponding to the step 2. According to data listed in table 1,

Table 1: test inputs and their corresponding result

id	test inputs	result
1	(7, 2, 4, 3)	PASS
2	(7, 2, 4, 5)	Ex 2
3	(7, 2, 6, 3)	PASS
4	(7, 2, 6, 5)	PASS
5	(7, 4, 4, 3)	Ex 1
6	(7, 4, 4, 5)	Ex 1
7	(7, 4, 6, 3)	PASS
8	(7, 4, 6, 5)	PASS
9	(7, 5, 4, 3)	PASS
10	(7, 5, 4, 5)	Ex 2
11	(7, 5, 6, 3)	PASS
12	(7, 5, 6, 5)	PASS
13	(11, 2, 4, 3)	PASS
14	(11, 2, 4, 5)	Ex 2
15	(11, 2, 6, 3)	PASS
16	(11, 2, 6, 5)	PASS
17	(11, 4, 4, 3)	Ex 1
18	(11, 4, 4, 5)	Ex 1
19	(11, 4, 6, 3)	PASS
20	(11, 4, 6, 5)	PASS
21	(11, 5, 4, 3)	PASS
22	(11, 5, 4, 5)	Ex 2
23	(11, 5, 6, 3)	PASS
24	(11, 5, 6, 5)	PASS

Table 2: Identified MFSs and their corresponding Exception

MFS	Exception
(-, 4, 4, -)	Ex 1
(-, 2, 4, 5)	Ex 2
(-, 3, 4, 5)	Ex 2

we can deduce that that (-, 4 , 4, -) must be the MFS of Ex 1 as all the inputs triggered Ex 1 contain this schema. Similarly, the schema (-, 2, 4, 5) and (-, 3, 4, 5) must be the MFSs of the Ex 2. We listed the MFSs and its corresponding exception in table 2.

Note that we didn't get the expected result with traditional MFS identifying approaches for this case. The MFSs we get for Ex 2 are (-,2,4,5) and (-,3,4,5) respectively instead of the expected schema (-,-,4,5). So why we can't identify the MFS (-,-,4,5)? The reason lies in the two inputs: input 6. (7,4,4,5) and input 18. (11,4,4,5). This two inputs contain the schema (-,-,4,5), but didn't trigger the Ex 1, instead, the Ex 2 was triggered.

Now let us get back to the source code of *foo*, we can find that if Ex 1 are triggered, it will stop executing the remaining code and report the exception information. In another word, Ex 1 have a higher level than Ex 2 so that Ex 1 may mask Ex 2. With this information, we can suppose that for the input (7,4,4,5) and (11,4,4,5), Ex 1 may masked Ex 2. Then we exam the schema (-,-,4,5), we can find all the test inputs contain (-,-,4,5) will trigger exception 1, except these test cases trigger Ex 2 first. So we can conclude that (-,-,4,5) should be the causing schema of the Exception 1. So the MFS information will be updated to table 3 which is identical to the expected result.

Table 3: expected MFSs and their corresponding Exception

MFS	Exception
(-, 4, 4, -)	Ex 2
(-, -, 4, 5)	Ex 1

So in this paper, we need to analysis the priority among the faults in the SUT and use the information to assist the MFS identifying algorithms to make the result more accurate and clearer.

3. PRELIMINARY

Before we talk about our approach, we will give some formal definitions and background first, which is helpful to understand the description of our approach.

3.1 Combinatorial testing

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

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

Definition 2. We consider the fact that abnormal executing of the SUT as a *fault*. It can be a exception, a compilation error, a mismatched assertion or a constraint violation.

Definition 3. The *priority* is a function indicate the priority relationship between two faults. Specifically, we take $Priority(F_a, F_b) = 1$ as that fault F_a has a higher level than F_b, which means that if F_a were triggered, it will omit the code that may trigger F_b. And $Priority(F_a, F_b) = -1$ indicate that F_a has a lower level than F_b. Finally, we take $Priority(F_a, F_b) = 0$ as that there is no priority relationship between F_a and F_b, in another word, neither F_a will mask F_b nor F_b will mask F_a.

Definition 4. For the SUT, the n-tuple (-,v_{n_1},...,v_{n_k},...)is called a k-value *schema* (k > 0) when some k parameters have fixed values and the others can take on their respective allowable values, represented as "-". In effect a test configuration its self is a k-value *schema*, which k is equal to n. Furthermore, if a test configuration contain a *schema*, i.e., every fixed value in this schema is also in this test configuration, we say this configuration hit this *schema*.

Definition 5. let s_l be a l-value schema, s_m be an m-value schema for the SUT and $l \leq m$. If all the fixed parameter values in s_l are also in s_m, then s_m *subsumes* s_l. In this case we can also say that s_l is a *sub-schema* of s_m and s_m is a *parent-schema* of s_l.

Definition 6. If all test configurations except these configurations triggered a higher level fault contain a schema, say S_a, trigger a particular fault, say F_a, then we call this schema S_a the *faulty schema* for F_a. Additionally, if none sub-schemas of S_a is the *faulty schema* for F_a, we will call the schema S_a the *minimal faulty schema* for F_a(*MFS* for short).

Note that, traditional MFS definition didn't consider the priority relationship among faults, so these definition will not take the schema as a MFS for some particular fault if some test configuration contain this schema doesn't trigger this fault.

The target of traditional MFS identifying algorithms is to find thes MFSs for the faults of a SUT. For by doing that can help the developers reduce the scope of source code that needed to inspect to debug the software. Note that our discuss is based on the SUT is a deterministic software, i.e., SUT execute under a test configuration will not pass one time and fail another time. The non-deterministic problem will complex our test scenario, which, however is beyond the scope of this paper.

fault. level. given this definition, we can also take the constraint as a fault, usually it will have the highest level, for that if we take a combination which is constraint, it will may didn't compile at all, so that we can't exam any code in this software.

test configuration?

schema. faulty, healthy. we need to identify the minimal faulty schema, that is MFS.

note that in our paper, we just consider the failure that is deterministic.

4. THEORY FOUNDATION

For a SUT, assume it has n different faults: $F_i(1 \leq i \leq L)$, they can be distinguished by the fault information come with them(such as exception traces or fatal code). Without loss of generality, we assume monotonicity of the level of faults ($Level(F_1) \geq Level(F_2) \geq ... \geq Level(F_L)$).

Let S_t denote all the schemas in the test configuration t. For example, If T = (1,1,1). Then S_T is {(1,1,1),(1,1,-),(1,-,1),(-,1,1),(-,-,1),(-,1,-)}.

Similarly, Let S_T denote all the schemas in a suite of test configurations – T. In fact, $S_T = \bigcup_{i=1}^{n} S_{t_i}$.

The notation T_{F_i} denote a suite of test configurations that each test configuration in it will trigger fault F_i. Additionally, let T_P denote the suite of test configurations that passed the testing without triggering any faults, and let T_F indicate the suite of test configurations that each one in this set will trigger some type of fault. It is obviously that $T_F = \bigcup_{i=1}^{L} T_{F_i}$ and $T_F \bigcup T_P = T_{all}$, In which, T_{all} denote all the possible test configurations that can generate to test the SUT.

Then let $M_{F_i}(1 \leq i \leq L)$ denote the set of MFS for F_i. It is noted that for F_i there may be more than one MFS, so we use set of MFS instead one MFS for a fault.

As MFS is also a schema, so we can easily find the followed properties: $M_{F_i} \in S_{T_{F_i}}$

4.1 Regard as one fault

In our previous work(TOSEM), we didn't distinguish the types of fault, i.e., we treat all the faults as one failure. In this circumstance, our MFS identifying process can be formulated as:

$$M_F = \{s_i | s_i \in S_{T_F} - S_{T_P} \ and \ \nexists s_j \prec s_i \ s.t. \ s_j \in S_{T_F} - S_{T_P}\}$$

With this formula we cannot find the MFS for a particular fault, further more, as it force to merge all the MFS of, it may get a wrongly result. For example, for SUT(3,2), Assume (0,-,0) and (-,0,0) are the MFS of err 1 and (1,1,-) and (1,-,0) is the MFS of err 2. Err 1's level is higher than Err 2. The test configurations and result is listed in table 4.

Table 4: a simple example

id	test inputs	result
1	(0, 0, 0)	Err 1
2	(0, 0, 1)	PASS
3	(0, 1, 0)	Err 1
4	(0, 1, 1)	PASS
5	(1, 0, 0)	Err 1
6	(1, 0, 1)	PASS
7	(1, 1, 0)	Err 2
8	(1, 1, 1)	Err 2

Table 5: previous work

id	test inputs	result
1	(0, 0, 0)	FAIL
2	(0, 0, 1)	PASS
3	(0, 1, 0)	FAIL
4	(0, 1, 1)	PASS
5	(1, 0, 0)	FAIL
6	(1, 0, 1)	PASS
7	(1, 1, 0)	FAIL
8	(1, 1, 1)	FAIL

Then using our previous work, firstly, we will treat the test result as listed in 5.

And then we will wrongly identified as (1,1,-) and (-,-,0) for the fault.

4.2 distinguish fault

To solve this problem, a naturally solution is to distinguish these faults, and for a particular fault, say F_i, we treat the test configurations trigger this fault as failure test configuration and the remained test configurations (consist of pass and other faults test configurations) will be regarded as pass.

So for this idea, the identifying process for a particular fault, say, F_i can be formulated as follows:

Let

$$S_{ca} = S_{T_{F_i}} - S_{T_P} - \bigcup_{j=1 \& j \neq i}^{L} S_{T_{F_j}}$$

$$M_F = \{s_i | s_i \in S_{ca} \ and \ \nexists s_j \prec s_i \ s.t. \ s_j \in S_{ca}\}$$

Using this we will treat the table 4 as the following table 6, then we will get the result as : (0,-,0) and (-,0,0) are the MFS of err 1 and (1,1,-) is the MFS of err 2. It is very approximate to the solution except that it loose the (1,-,0) for err 2. This is because in fact when we look at the 5th test configuration (1,0,0), it has already triggered the err 1 which have a higher level than err 2 so that err 2 is not triggered. But this approach just regard the 5th test configuration as a passed test configuration as it does not trigger err 2.

4.3 Test configurations with higher level fault masked every lower level fault

So what if we just consider the configurations triggered higher fault as having triggered the lower fault, in other

Table 6: do not consider the masking effect

ERR 1			ERR 2		
id	test inputs	result	id	test inputs	result
1	(0, 0, 0)	FAIL	1	(0, 0, 0)	PASS
2	(0, 0, 1)	PASS	2	(0, 0, 1)	PASS
3	(0, 1, 0)	FAIL	3	(0, 1, 0)	PASS
4	(0, 1, 1)	PASS	4	(0, 1, 1)	PASS
5	(1, 0, 0)	FAIL	5	(1, 0, 0)	PASS
6	(1, 0, 1)	PASS	6	(1, 0, 1)	PASS
7	(1, 1, 0)	PASS	7	(1, 1, 0)	FAIL
8	(1, 1, 1)	PASS	8	(1, 1, 1)	FAIL

Table 7: over mask example

ERR 1			ERR 2		
id	test inputs	result	id	test inputs	result
1	(0, 0, 0)	FAIL	1	(0, 0, 0)	FAIL
2	(0, 0, 1)	PASS	2	(0, 0, 1)	PASS
3	(0, 1, 0)	FAIL	3	(0, 1, 0)	FAIL
4	(0, 1, 1)	PASS	4	(0, 1, 1)	PASS
5	(1, 0, 0)	FAIL	5	(1, 0, 0)	FAIL
6	(1, 0, 1)	PASS	6	(1, 0, 1)	PASS
7	(1, 1, 0)	PASS	7	(1, 1, 0)	FAIL
8	(1, 1, 1)	PASS	8	(1, 1, 1)	FAIL

words, we defaulted think that for each test configurations triggering a fault, say, F_i, it has masked all the faults has a level lower than it, i.e., these faults $F_j, i < j \leq L$ will be triggered in these configurations if the fault F_i does not trigger. For this idea, the identifying process for a particular fault with considering masking effect, say, F_i can be formulated as follows:

Let

$$S_{ca} = \bigcup_{j=1}^{i} S_{T_{F_j}} - S_{T_P} - \bigcup_{j=i+1}^{L} S_{T_{F_j}}$$

$$M_F = \{s_i | s_i \in S_{ca} \ and \ \nexists s_j \prec s_i \ s.t. \ s_j \in S_{ca}\}$$

We can see the different part with the previous formula is that it just minus the schemas in these passing test configurations and these test configurations triggering a lower level than F_i. With this, we will treat the table 4 as : table 7. This time we get (0,-,0) and are the MFS of err 1, (1,1,-) and (-,-,0) for the err 2. Obviously it is still not the correct answer.

This is because we look at 1th, 3th test configuration, actually they have trigger the err 1 which has the higher level than err 2, but it does not mask the err 2 as this two test configurations didn't contain the MFS for err 2 we set at first. So they should be regard as pass test configuration for err 2, but this approach didn't know this, they just set all the test configurations that trigger higher level fault as the fail test configuration for the under test fault.

4.4 Ideal solution

To get an ideal solution, we should make the followed assumption:

For a particular fault, say, F_i, with ,these test configurations triggered higher fault are $\bigcup_{j=1}^{i-1} T_{F_j}$ .Assume we have known previously that in this set some test configurations will trigger F_i if the higher fault will not triggered, we label this set as

T_{tri-F_i} (this part is needed because of if there are more than two faults in a SUT, the higher fault may even make the lower fault don't appear)

And some test configurations in this set will not trigger F_i. We label them as:

$T_{\neg tri-F_i}$.

Obviously,$T_{tri-F_i} \bigcup T_{\neg tri-F_i} = \bigcup_{j=1}^{i-1} T_{F_j}$.

At last, we should do as the following to get the ideal computing formula:

Let

$$S_{ca} = S_{T_{tri-F_i}} + S_{T_{F_i}} - S_{T_P} - S_{T_{\neg tri-F_i}} - \bigcup_{j=i+1}^{L} S_{T_{F_j}}$$

$$M_F = \{s_i | s_i \in S_{ca} \ and \ \nexists s_j \prec s_i \ s.t. \ s_j \in S_{ca}\}$$

So still for the example table 4, we will first get $T_{tri-F_i} = \{(1,0,0)\}$ and $T_{tri-F_i} =$ (0, 0, 0), (0, 1, 0)

And then we will treat table 4 as table 8, so this time we will accurately get the expected result:

(0,-,0) and (-,0,0) are the MFS of err 1 and (1,1,-) and (1,-,0) is the MFS of err 2.

Table 8: ideal solution

ERR 1			ERR 2		
id	test inputs	result	id	test inputs	result
1	(0, 0, 0)	FAIL	1	(0, 0, 0)	PASS
2	(0, 0, 1)	PASS	2	(0, 0, 1)	PASS
3	(0, 1, 0)	FAIL	3	(0, 1, 0)	PASS
4	(0, 1, 1)	PASS	4	(0, 1, 1)	PASS
5	(1, 0, 0)	FAIL	5	(1, 0, 0)	FAIL
6	(1, 0, 1)	PASS	6	(1, 0, 1)	PASS
7	(1, 1, 0)	PASS	7	(1, 1, 0)	FAIL
8	(1, 1, 1)	PASS	8	(1, 1, 1)	FAIL

4.5 limitations in practice

In practice, we cannot use the ideal formula to compute the MFS for each fault for there are three main limitations which we will encounter:

1. We cannot execute all the test configurations if the parameters and their values it too much for the SUT, that is in practice, $T_F \bigcup T_P \neq T_{all}$
2. If we find the faults, we can't judge the levels of these faults just using black-box testing method.
3. Even if we have known the levels of each faults, we still cannot decide the value of $S_{T_{tri-F_i}}$ and $S_{T_{\neg tri-F_i}}$ without fixing the higher level fault than F_i and re-executing the test configurations.

5. APPROACH DESCRIPTION

As we cannot get an ideal solution because of the three limitations proposed in the previous section, our target then is to find a practical solution to improve the efficiency for the existing MFS identifying algorithms when facing multiple faults , i.e.,lowering variance of identifying MFS in a SUT with multiple faults as much as possible.

To get the target, first let's get back to the traditional MFS identifying algorithms to see how they works. In fact, they all can be represent as the following formula:

Let

$$S_{ca} = S_{T'_{F_i}} - S_{T'_P}$$

$$M_F = \{s_i | s_i \in S_{ca} \ and \ \nexists s_j \prec s_i \ s.t. \ s_j \in S_{ca}\}$$

A noted point is that $T'_{F_i} \subseteq T_{F_i}$ and $T'_P \subseteq T_P$ as that we can't execute all the possible test configurations in a SUT in practice when the scale of the configuration space is big.

The difference among these algorithms are just at T'_{F_i} and T'_P. However, what the difference in detail is not the point in this paper. We should also note that when the SUT just have one fault, these algorithms all can get a good result. So what we want to do is to improve these algorithms when the SUT can have multiple faults.

As we have mentioned in the previous section, the $S_{T'_{F_i}} - S_{T'_P}$ is not completed. This is because there may be some failure-inducing schemas in some T_{F_j} we did not add and there may be some healthy schemas in T_{F_j} we did not minus. This two factors we can't improve, however, because we neither know the levels of these faults nor know value of $S_{T_{tri-F_i}}$ and $S_{T_{\neg tri-F_i}}$. So what we can do is just to increase the number of T'_{F_i} and T'_P to increase the accurately of identifying the MFS of a particular fault F_i.

5.1 Replace test configuration that trigger unexpected fault

The basic idea is to discard the test configurations that trigger other faults and generate other test configurations to represent them. These regenerate test configurations should either pass the executing or trigger F_i. The replacement must fulfil some criteria, such as for CTA, the input for this algorithm is a covering array, and if we replace some test configuration in it, we should ensure that the covering rate is not changed.

Commonly, when we replace the test configuration that trigger unexpected fault with a new test configuration, we should keep some part in the original test configuration, we call this part as *fixed part*, and mutant other part with different values from the original one. For example, if a test configuration (1,1,1,1) triggered Err 2, which is not the expected Err 1, and the fixed part is (-,-,1,1), then we may regenerate a test configuration (0,0,1,1) which will pass or trigger Err 1.

The *fixed part* can be the schemas that only appear in the original test configuration which other test configurations in the covering array did not contain(for the algorithm take covering array as the input: CTA), or the factors that should not be changed in the OFOT algorithms, or the part that should not be mutant of the test configuration in the last iteration(FIC_BS).

The process of replace a test configuration with a new one with keeping some fixed part is depicted in Algorithm 1:

The inputs for this algorithm consists of a test configuration which trigger an unexpected fault – $t_{original}$, the fixed part which we want to keep from the original test configuration – s_{fixed} , the fault type which we current focus – F_i. And the values sets that each option can take from respectively. The output of this algorithm is a test configuration t_{new} which either trigger the expected F_i or passed.

Algorithm 1 replace test configurations that trigger unexpected fault

Input: $t_{original}$ ▷ original test configurations
F_i ▷ fault type
s_{fixed} ▷ fixed part
$Param$ ▷ values that each option can take
Output: t_{new} ▷ the regenerate test configuration
1: **while not** MeetEndCriteria() **do**
2: $s_{mutant} \leftarrow t_{original} - s_{fixed}$
3: **for each** $opt \in s_{mutant}$ **do**
4: $i = getIndex(Param, opt)$
5: $opt \leftarrow opt' \ s.t. \ o \in Param[i] \ and \ opt' ! = opt$
6: **end for**
7: $t_{new} \leftarrow s_{fixed} \bigcup s_{mutant}$
8: $result \leftarrow execute(t_{new})$
9: **if** $result == PASS \ or \ result == F_i$ **then**
10: **return** t_{new}
11: **else**
12: continue
13: **end if**
14: **end while**
15: **return** *null*

In fact, this algorithm is a big loop(line 1 - 14) which be parted into two parts:

The first part(line 2 - line 7): generate a new test configuration which is different from the original one. This test configuration will keep the fixed part (line 7), and just mutant these factors are not in the fixed part(line 2). The mutant for each factor is just choose one legal value that is different from the original one(line 3 - 6). The choosing process is just by random and the generated test configuration must be different each iteration(can be implemented by hashing method).

Second part is to validate whether this newly generated test configuration matches our expect(line 8 - lone 13). First we will execute the SUT under the newly generated test configuration(line 8), and then check the executed result, either passed or trigger the expected fault – F_i will match our expect.(line 9) If so we will directly return this test configuration(line 10). Otherwise, we will repeat the process(generate newly test configuration and check)(line 11 -12).

It is noted that the loop have another end exit besides we have find a expected test configuration(line 10), which is when function *MeetEndCriteria()* return a true value(line 1). We didn't explicitly show what the function *MeetEndCriteria()* is like, because this is depending the computing resource you own and the how accurate you want to the identifying result to be. In detail, if you want to your identify process be more accurate and you have enough computing resource, you can try much times to get the expected test configuration, otherwise, you may just try a relatively small times to get the expected test configuration.

In this paper, we just set 3 as the biggest repeat times for function. When it ended with *MeetEndCriteria()* is true, we will return null(line 15), which means we cannot find a expected test configuration.

5.2 Examples when apply this approach into some MFS identifying algorithms

Next we will take three MFS identifying algorithms as the subject to see how our approach works on them.

Table 9: Identifying MFS using OFOT with our approach

original test configuration	fault info
0 0 0 0	Err 1
gen test configurations	**result**
1 0 0 0	Err 1
~~0 1 0 0~~	~~Err 2~~
0 2 0 0	Err 1
0 0 1 0	Pass
0 0 0 1	Pass
original identified:	**identified with replacement**
(- 0 0 0)	(- - 0 0)

5.2.1 OFOT examples

Assume we have test a system with four parameters, each has three options. And we take the test configuration (0 0 0 0) we find the system encounter a failure called "Err 1". Next we will take the MFS identifying algorithms – OFOT with the help of our approach to identify the MFS for the "Err 1". The process is listed in table 9. In this table, The test configuration which are labeled with a deleted line represent the original test configuration generated by OFOT, and it will be replaced by the regenerated test configuration which are labeled with a wave line under it.

From this table, we can find the algorithm mutant one factor to take the different value from the original test configuration on time. Originally if the test configuration encounter the different condition with the Err 1, OFOT will make a judgement that the MFS was broken, in another word, if we change one factor and it does not trigger the same fault, we will label them as one failure-inducing factor, after we changed all the elements, we will get the failure-inducing schemas. For this case, as when we change the second factor , third factor and the fourth factor, it doesn't trigger the Err 1 (for second factor, it trigger Err 2 and for the third and fourth, it passed). So if we do not regenerate the test configuration (0 2 0 0), we will get the MFS – (- 0 0 0) for the err 1(which are also labeled with a delete line).

However, if we replace the test configuration (0 1 0 0) with (0 2 0 0) which triggered err 1 (in this case, the fixed part of the test configuration is (0, - - -)), we will find that only when we change the third and fourth factor will we broke the MFS for err 1, so with our approach, we will find the MFS for err 1 should be (- - 0 0) (labeled with a wave line under it).

5.2.2 CTA examples

CTA uses the covering array as its inputs and then use classification tree algorithm to characterize the MFS. For this algorithm, we still assume the SUT has 4 parameters and each one has 3 values. Then we will initial a 2-way covering array as the input for CTA algorithm which is listed in table 10. The delete line and wave line have the same meaning as the OFOT example.

Let's look at the original test configuration (0 1 1 1), it triggered the err 2 which is not as our expected, so we will replace it with other test configurations. A prerequisite for

Table 10: Identifying MFS using CTA with our approach

covering arrays	fault info
0 0 0 0	Err 1
~~0 1 1 1~~	~~Err 2~~
1 1 1 1	Err 1
0 1 1 0	Pass
0 0 0 1	Pass
0 2 2 2	Pass
1 0 1 2	Pass
1 1 2 0	Err 1
1 2 0 1	Err 1
2 0 2 1	Pass
2 1 0 2	Pass
2 2 1 0	Pass
original identified:	**identified with replacement**
(0 0 - -) : err1	(0 0 0 -):err1
(1 1 - -) : err1	(1 1 - -):err1
(1 2 - -) : err1	(1 2 - -):err1
(0 1 - -) : err2	

this replacement is that we should not decrease the covering rate. As the original test configuration contain the 2-degree schema(0 1 - -),(0 - 1 -),(0 - - 1),(-,1,1,-),(-,1,-,1),(-,-,1,1). We make them as fixed part that the regenerate test configuration must keep, as we cannot use one test configuration to cover all the six fixed part, so instead, we use three additional test configurations (1 1 1 1), (0, 1 ,1 ,0) and (0,0,0,1) to cover them, note that this three test configurations either trigger err 1 or pass. If they don't match this condition, we will try other test configurations to replace the original test configuration.

We use multiple-class CTA as our subject, Ylimaz in the paper has claimed that mutiple-class CTA perferom better than terrncy-class CTA.

5.2.3 FIC_BS examples

FIC adaptively generate test configurations according to the executing result of last test configuration. So we will observe each time it generate a test configuration to see whether should we replace the newly generWe observe that, for most cases, augment approaches have an promotion against traditional ones. In fact, there are 10 cases out of 14 outperform than traditional ones at the metric of similarity, and 9 out of 14 better than traditional ones at the metric of num diff. The extent is distingushing, the best is (0.9) , the average performance reach (+ 0.6)

So the answer for Q3 is: our approach do get a better performance at identifying failure-inducing combinations when facing masking effect between multiple faults, to which the extent is not less.ated test configuration. Take the following example, that a SUT contain 8 parameters and each parameter has three values, we set the number of parameters to be 8 because it can get a better description of this algorithm.

As which is last test configuration will have a significance impact for FIC, so if we replace one test configuration during the process of the FIC_BS, the generated test configurations will have a big difference from traditional FIC_BS. To clearly describe the influence of our approach on FIC_BS, we will first give a completed example of the traditional identifying process of FIC_BS, and then give the process with our approach. The detailed is listed in table 11.

For the first part, we can easily learn that FIC_BS use a binary search strategy to mutant the parameters in a test configuration. And which part will be mutant is based on the executing result of last test configuration. We are not going to describe the FIC_BS algorithms in detail. Instead, we just focus on the test configuration (1 1 1 0 0 0 0 0) triggering err 2. As traditional FIC_BS will regard it as a passing test configuration, together with results of previous test configurations, it will judge that (- - 0 - - - - -) must be a failure-inducing factor. And finally it will take (- - 0 0 - 0 - -) as the MFS for err 1.

However, when apply our approach, we will replace the test configuration (1 1 1 0 0 0 0 0) to be (2 2 2 0 0 0 0 0), ((- - - 0 0 0 0 0) is thReplacee fixed part). And then we will find it still trigger the err 1, which means (- - - 0 - - - -) should be failure-inducing factor rather than (- - 0 - - - - -) in the traditional approach. And this step will have a influence on the test configurations generated following, and finally we will identify the schema (- - - 0 - 0 - -) as the MFS for err 1.

Table 11: Identifying MFS using FIC_BS

original test configuration	**fault info**
0 0 0 0 0 0 0 0	Err 1
Use traditional FIC_BS	
gen test configurations	**result**
1 1 1 1 0 0 0 0	PASS
1 1 0 0 0 0 0 0	Err 1
1 1 1 0 0 0 0 0	Err 2
1 1 0 1 1 1 1 1	PASS
1 1 0 1 1 0 0 0	PASS
1 1 0 1 0 0 0 0	PASS
1 1 0 0 0 0 0 0	Err 1
1 1 0 0 1 1 1 1	PASS
1 1 0 0 1 1 0 0	PASS
1 1 0 0 1 0 0 0	Err 1
1 1 0 0 1 0 1 1	Err 1
Mfs identified	
(- - 0 0 - 0 - -)	for Err 1
Use FIC_BS with our approach	
gen test configurations	**result**
1 1 1 1 0 0 0 0	PASS
1 1 0 0 0 0 0 0	Err 1
~~1 1 1 0 0 0 0 0~~	~~Err 2~~
2 2 2 0 0 0 0 0	Err 1
1 1 1 0 1 1 1 1	PASS
1 1 1 0 1 1 0 0	PASS
1 1 1 0 1 0 0 0	Err 1
1 1 1 0 1 0 1 1	Err 1
Mfs identified	
(- - - 0 - 0 - -)	for Err 1

6. EMPIRICAL STUDIES

We conducted several case studies to address the following questions:

Q1: Do masking effects existed in real software when it contain multiple faults?

Q2: What is the extent to which traditional approaches suffer from these real masking effects?

Q3: Can our approach do better than traditional approaches when facing these masking effects?

Specifically, section 6.1 survey several open-source software to gain a insight of the state of the existence of multiple faults and their masking effects. Section 6.2 directly applied three MFS-identifying programs on the surveyed software and analysis their results. Section 6.3 apply our approach on the software and a comparison with traditional approaches will be discussed. Section 6.4 discuss the threats to validity of our empirical studies.

6.1 study 1: multiple faults and masking effects in practice

In the first study, we surveyed several software to gain a insight of the state of the existence of multiple faults and their effects. The software under study are four GNU software: Grep, Sed, Make, Gzip and a database management software- HSQLDB. Each of them contain different versions. All the three subjects are highly configurable so that the options and their combination can influence their behaviour. Additionally, they all have developers' community so that we can easily get the real bugs reported in the bug tracker forum. Table 12 lists the program, the number of versions we surveyed, number of lines of uncommented code, number of procedures(for c software) or number of classes (for java software) and a brief description.

6.1.1 study setup

Table 12: Software under survey

software	versions	LOC	procedur/classes	description
Grep	5	9493-10102	146	a program to search for strings inside a file.
Sed	7	5503-14477	255	A stream editor that parses and transforms text
Make	4	13359-35583	268	a utility that automatically builds executable programs and libraries.
Gzip	5	4604-5754	104	a popular data compression program
HSQLDB	2	-	-	A database management software written with pure java

Table 13: number of faults and their masking effect condition

software	versions	Faults ID	all tests	Num masking
HSQLDB	2cr8	#2000 & #3000	2000	1000

We first looked through the bug tracker forum of each software and scratched some bugs which are caused by the options combination to study later. We then classify these bugs according to the version they belong to. For each bug, we will derive its failure-inducing combinations by analysing the bug description report and its attached test file which can reproduce the bug. For example, through analysing the source code of the test file of bug#123213 for HSQLDB, we found the failure-inducing combinations for this bug is: (*preparestatement*, *placeHolder*, *Long string*), this three factors together form the condition on which the bug will be triggered. We call these analysed result the "perfect combinations" which will be used as the comparison benchmark in the later studies.

We further selected pairs of bugs belong to the same version and merged their test file. This merging manipulation vary with the pair of bugs we selected. As for bug#123213 and bug#223012 of HSQLDB, the original test file can be simply described as figure 1:, and we merged them together into a file like figure 2, we can easily find that it use a test scenario to merge the two test file. For each pair of bugs, we have posted the source code of the merging file on website.

Next we built the input model which consist of the options related to the perfect combinations and additional noise options. The detailed model information is listed in appendix. We then generated the exhaustive test suite consist of all the possible combinations of these options under which we executed the merged test file. We record the output of each test case to observe whether there are test cases contain 'perfect combination' but do not produce the corresponding bug.

6.1.2 result and discuss

Table 13 lists the results of our survey. Column "Faults ID" indicate the IDs of two bugs we collected from the bug tracker system, column "all tests" give the total number of test cases we executed and column "num of masking" indicate the number of test cases which trigger the masking effect.

We observed that for each pair of bugs we listed in the table, we can always find some test cases, although contain perfect combinations, did not trigger expected bug, i.e., the expected bug was masked. Specifically, there are 1000 out of 2000 (20%) test cases for HSQLDB triggered the masking effect.

So the answer to Q1 is that in practice, when SUT have multiple faults, there is a good chance that masking effect can be triggered by some test cases.

6.2 study 2:performance of traditional algorithms

In the second study, we want to learn how badly the masking effect impact on the traditional approaches. To conduct this study, we need to apply the traditional failure-inducing identifying algorithms on the SUT we collected in the first study and compare them with the perfect combinations.

6.2.1 study setup

The traditional approaches we selected are: OFOT, FIC and CTA. To make the first two approaches (OFOT and FIC) work, we need to feed them with a failing test case. And for CTA, however, it need to work with a covering array and the executing result for each test case in the covering array. As the input for the traditional approaches are different, we designed two different setups, one for OFOT and FIC, while the other for CTA.

For OFOT and FIC, our set up is as follows:

We first selected failing test cases from the exhaustive executed test cases, and for each test case, we applied OFOT and FIC to isolate the failure-inducing combinations in this test case. As the subject under test has multiple faults, the strategy we chose for this two the traditional approaches is *distinguish fault* in section 4.2, i.e., distinguish fault and ignore the masking effect happened among them. At last we collected all the failure-inducing combinations they got and deleted those overlapped ones.

For CTA, our setup is as follows:

We first utilize augment simulating algorithms(ASA) to generate 2-way covering arrays. And for each test case in the covering array, the executing result of them can be fetched from the exhaustive set we have got in the first study. Then we fed CTA approach with the covering array along with the corresponding executing results. We then collected the failure-inducing combinations after running CTA. As different covering array may affect the result of CTA approach, we repeated using ASA to generate 2-way covering array for 30 times and applied CTA for each of them. It is noted that ASA algorithm is a heuristic approach, which containing some random factors, so the 30 covering arrays are different from each other.

After we have collected the result got by each algorithm for each version software, we next need to compare the result with the perfect combinations to quantify the extent to which traditional approaches suffers from masking effect. To make this comparison, we first introduce the following notation:

Table 14: traditional result

Software	version	num diff			similarity		
-	-	OFOT	FIC	CTA	OFOT	FIC	CTA
HSQLDB	3.14	3	4	4	0.65	0.64	0.5

Table 15: our approach result

Software	ver	num diff			similarity	
-	-	OFOT	FIC	CTA	OFOT	FIC
HSQLDB	3.14	3 (+4)	4 (+4)	4 (-2)	0.65 (+0.01)	0.64(-0.05)

Assume we get two schema S_A,S_B, the notation$S_A \bigcap S_B$ indicate the same elements between S_A and S_B. For example $(-1\ 2 - 3) \bigcap (-2\ 2 - 3) = \{(- - 2 - -), (- - - - 3)\}$.

Then the similarity between schemas is defined as the followed notation:

$$Similarity(S_A, S_B) = \frac{|S_A \bigcap S_B|}{\max(Degree(S_A), Degree(S_B))}$$

.

In this formula, the numerator indicate the number of same elements in S_A and S_B. It is obviously, the bigger the value is the more similar two schemas are. The denominator give us this information: if the the number of same elements is fixed, the bigger the degree of the schema, the more noise we will encounter to find the fault source. In a word, the bigger the formula can get the better the similarity is between the two schema.

For the set of schemasSet_A and Set_B, the similarity definition is :

$$Similar(Set_A, Set_B) = \frac{\sum_{s_i \in Set_A} \max_{s_j \in Set_B} S(s_i, s_j)}{|Set_A|}$$

.

It is obviously the more similarity between the set of failure-inducing combinations identified by the algorithms with these perfect combinations the less impacts do these algorithms suffered from multiple faults as well as their masking effects.

Besides the similarity metric, another metric also need to be considered: the discrepancy between the number of identified failure-inducing combinations with these perfect combinations. If this discrepancy is too big, even though we get a good similarity metric value, we also get too many noise which make our result inaccuracy.

Particularly, if the value of discrepancy number metric is 0 and the value of similarity metric is 1, it means that multiple faults as well as their masking effect have no impacts on the algorithms, which is usually not feasible in practice.

6.2.2 result and discuss

Table 14 depicts the result of the second case study. Column "Num Diff" indicates the discrepancy of the number of MFS between the ones got through traditional approaches with the perfect combinations. Column "Similarity" presents the similarity between this two set of combinations. Additionally, the results of different algorithms can be distinguished by the titles of sub-columns, which are "FIC","OFOT","CTA", indicating corresponding algorithms respectively. It is noted that for column "CTA", the vale we set in each cell are the average value of the result we collected from 30 repeated experiments' result for a particular subject.

From this table, we can easily find that Traditional MFS identifying approaches do suffer from the multiple faults and their masking effect. For example, for the Grep, version 3.14, all these three algorithm will lost some information. Additionally, the extent to which the impact has affected vary from algorithms. We can find that for HSQLDB with version 3.14, the similarity of CTA was just at 4 while FIC is 3. We list these points in figure 3. From this figure, we can learn that.

So the answer for Q2 is: traditional algorithm do suffer from the multiple faults and their masking effect although the extent vary in different algorithms.

6.3 study 3:performance of our approach

The last case study aims to observe the performance of our approach and compare it with the result got by the traditional approaches. Specifically, we augmented the three traditional approaches using the method described in section 5, and then we applied these augmented approaches on identifying the failure-inducing combinations in the prepared subjects.

6.3.1 study setup

The setup of this case study is almost the same as the second case study. The difference is that the algorithms we choose are three augment ones. Additionally, comparisons between augment approaches with three traditional ones will be quantified.

6.3.2 result and analysis

Table 15 presents the result of the last case study. The form of this table is similar to the second study. We just added some information in the parentheses attached the value in each cell. This information display the discrepancy between traditional ones. Notation '+' means promotions against traditional ones, while '-' indicates decrease. The value in parentheses has been normalized.

We observe that, for most cases, augment approaches got promotions against traditional ones. In fact, there are 10 out of 14 cases that augment ones outperform traditional ones at the metric of "similarity", and 9 out of 14 cases that augment ones are better than traditional ones at the metric of "num diff". Additional, this promotion is distinct. As we can see, the best promotion for similarity is 0.9 and the best promotion for num diff is 0.7. Furthermore, the average performance promotion for similarity reached '+ 0.6' and this value reached '+ 0.5' for the num diff metric.

So the answer for Q3 is: our approach do get better performance at identifying failure-inducing combinations when facing masking effect between multiple faults, to which the extent is distinct.

6.4 threats to validity

There are several threats to validity for these empirical studies. First, we have only surveyed five open-source software, four of which are medium-sized and one is large-sized. This may impact the generality of our observations. Although we believe it is quite possible a common phenomenon in most software that contain multiple faults which can mask each other, we need to investigate more software to support our conjecture. The second threat comes from the input model we built. As we focused on the options related to the perfect combinations and only augmented it with some noise options, there is a chance we will get different result if we choose other noise options. More different options needed to be opted to see whether our result is common or just appeared in some particular input model. The third threats is that we just observed three MFS identifying algorithms, further works needed to exam more MFS identifying algorithms to get a more general result.

7. RELATED WORKS

Nie?s approach in [3] and [6] first separates the faulty-possible tuples and healthy-possible tuples into two sets. Subsequently, by changing a parameter value at a time of the original test configuration, this approach generates extra test configurations. After executing the configurations, the approach converges by reducing the number of tuples in the faulty-possible sets.

Delta debugging [5] proposed by Zeller is an adaptive divide?and-conquer approach to locating interaction fault. It is very efficient and has been applied to real software environment. Zhang et al. [4] also proposed a similar approach that can identify the failure-inducing combinations that has no overlapped part efficiently.

Colbourn and McClary [7] proposed a non-adaptive method. Their approach extends the covering array to the locating array to detect and locate interaction faults. C. MartÃ■nez [8-9] proposed two adaptive algorithms. The first one needs safe value as their assumption and the second one remove the assumption when the number of values of each parameter is equal to 2. Their algorithms focus on identifying the faulty tuples that have no more than 2 parameters.

Ghandehari.etc [10] defines the suspiciousness of tuple and suspiciousness of the environment of a tuple. Based on this, they rank the possible tuples and generate the test configurations. Although their approach imposes minimal assumption, it does not ensure that the tuples ranked in the top are the faulty tuples.

Yilmaz [11] proposed a machine learning method to identify inducing combinations from a combinatorial testing set. They construct a classified tree to analyze the covering arrays and detect potential faulty combinations. Beside this, FouchÃl' [12] and Shakya [13] made some improvements in identifying failure-inducing combinations based on Yilmaz?s work.

We[] have proposed an approach that utilize the tuple relationship tree.

Besides works that focus on identifying the failure-inducing schemas in test cases. There are some work focus on working around the masking effects. Firstly Ylimaz proposed it in[], and using the feedback-driven approach to work around the masking effect. In specific, it first using CTA classify the possible MFS and then eliminate them and generate new test cases to detect possible masked interaction in the next iteration. And then in[] it extend its work. The difference between his work and our work is that the main focus of that work is to generate test cases that didn't omit some schemas that may be masked by other schemas. And our work is main focus on identifying the MFS and avoiding the masking effect.

His masking is defined these only appeared in the masking test cases. Our masking is that any fault is masking is masking. This different definitions shows the focus different, He focus on let test cases do test these combinations, while we focus that any masking will negatively affect the performance of failure-inducing combination identifying algorithms.

8. CONCLUSIONS

Multiple faults in SUT can bias the result of traditional MFS identifying approaches. In this paper, we theoretically analysis the impact with the multiple faults and its masking effect. Additionally, we have presented an approach that can alleviate this impact. We further surveyed a series of software to find that multiple faults is common sense and the masking can also appear. Based on the software we have surveyed, we compare our approach and traditional ones to support the claim that our approach can help testers to effeicently find the failure-inducing schemas with less suffers from the multiple faults and its masking effect.

9. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and what have you. In the present case, for example, the authors would like to thank Gerald Murray of ACM for his help in codifying this *Author's Guide* and the **.cls** and **.tex** files that it describes.

10. REFERENCES

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

[2] 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.

[3] L. S. G. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker. Identifying failure-inducing combinations in a combinatorial test set. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 370–379. IEEE, 2012.

[4] 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.

[5] 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.

[6] 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.

[7] C. Yilmaz, E. Dumlu, M. Cohen, and A. Porter. Reducing masking effects in combinatorial interaction testing: A feedback driven adaptive approach. 2013.

APPENDIX

A. MASKING FAULTS STATISTIC

Table 16: real faults detailed

software	faults ID	masking faults ID	Web site
Grep	#29537	#33080	http://savannah.gnu.org/bugs/?group=grep
-	#29537	#33080	-
Sed	-	-	-
Make	-	-	-
Gzip	-	-	-
HSQLDB	-	-	-

Table 17: input model of HSQLDB

SQL properties(TRUE/FALSE)	
sql.enforce_strict_size, sql.enforce_names,sql.enforce_refs, sql.enforce_size, sql.enforce_types, sql.enforce_tdc_delete, sql.enforce_tdc_update	
table properties	**values**
hsqldb.default_table_type	CACHED, MEMORY
hsqldb.tx	LOCKS, MVLOCKS, MVCC
hsqldb.tx_level	read_commited, SERIALIZABLE
hsqldb.tx_level	read_commited, SERIALIZABLE
Server properties	**values**
Server Type	SERVER, WEBSERVER, INPROCESS
existed form	MEM, FILE
Result Set properties	**values**
resultSetTypes	TYPE_FORWARD_ONLY,TYPE_SCROLL_INSENSITIVE, TYPE_SCROLL_SENSITIVE
resultSetConcurrencys	CONCUR_READ_ONLY,CONCUR_UPDATABLE
resultSetHoldabilitys	HOLD_CURSORS_OVER_COMMIT, CLOSE_CURSORS_AT_COMMIT
option in test script	**values**
StatementType	STATEMENT, PREPAREDSTATEMENT
constraints	**values**
StatementType	STATEMENT, PREPAREDSTATEMENT