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

Alvin Chain
Department of computing
Hong Kong Polytechnic University
Hong Kong
cstschan@comp.polyu.edu.hk

ABSTRACT

Minimal failure-inducing schema(MFS) is a important concept in combinatorial testing that indicate the failure-inducing interaction of parameters in the software under test(SUT). Identify the MFS can help developers quickly reduce the search space needed to find the buggy source. Many algorithms are proposed to find these MFSs in the failing test cases. In practice, however, we find that these algorithms cannot behave as expected in the condition that SUT has multiple faults with different levels. For if so, a fault with higher level may be triggered and leaving the code which will trigger the fault with lower level not executed. Thus we cannot observe the fault with lower level, as a result, we will omit the MFS that related to the fault with lower level. We call this a masking effect.

In this paper, we propose a framework which can help the algorithms to avoid this masking effect when identify MFSs in the test cases. In this framework, we first static analysis the data flow of the test cases. Second we record fault as well as the code lines related to this fault during executing test cases. Then we will determine the levels of different faults we triggered during testing through finding the relationships of the code lines of each fault. By doing so we can judge whether a masking effect is happened during the process of identifying MFS and avoiding them by generating new test case that exposing the fault with lower level.We have applied our framework into several algorithms which focus on identifying the MFS in test cases and empirically studied the framework on two widely used open source software. Our result of the studies shows that our framework can effectively reduce the influence of masking effect.

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

1. INTRODUCTION

With the increase of requirements for more features and customisable, modern software are designed to be configurable and modular. While it can make software portable and flexible, it also bring many challenges to the testers when testing them. The major one of these challenges is that we must make sure the components or options in the software coexistence with each other. As exhaustive testing each possible combination is not impractical when there are large amount of components or options in the SUT, we should choose some of them to test with considering the cost. There are many strategies for one to select the test configurations among all the possible configurations. Combinatorial testing is one of them that can reach the coverage of all the interactions of components with the number of component(we called interaction strength) not more than t.

Which criteria we should choose or how to generate these test cases is not the point in this paper, however, we will focus on the followed problem: if we find some test configurations failed during executing, which subset of the combination of component is the source of this failure? In another word, we want to identify the failure-inducing interactions of component rather than just detect them. Many works (as well as our previous work) are proposed to solve this problem. Most of these works focus on how to identify as more MFSs as possible while just generating small size of extra test configurations.

In our recent studies, however, we find these algorithms cannot behave as expected in some subject software. Through

*(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 LaTeX2ε and BibTeX* at `www.acm.org/eaddress.htm`

WOODSTOCK '97 El Paso, Texas USA

a deep analysis, we find that there are multiple faults with different levels in these subject. It means that when we set up a test configuration and execute the SUT to observe the result, the high level fault will trigger first and perturb we examining the code that may trigger the low level fault. As a result we will omit some options or component in this test configuration that may be the cause of the low level fault. We call this a masking effect which make the MFS identifying algorithms not able to work properly.

In this paper, we propose a approach that can assist these algorithms to avoid these masking effect. Our framework consists of three parts: first, it will use the statistic analysis technique–dominate tree to analysis the test script and then collect the information traditional identifying algorithms. of code lines in this script. Second, we will support a interface called "Record" for the MFS identifying algorithms that each time the algorithm encounter a fault should call this interface. So that we can record this fault as well as the code lines that trigger this fault. Last, this framework support these algorithms the interface "analysis" that can tell them whether the fault they encounter having masked some fault else.

To evaluate the effectiveness of our framework, we took two widely-used open source software as our experiment subject. And then we will choose five MFSs identifying algorithms, for each algorithm, we will compare the identifying result among two versions of this algorithm, one using our framework while another one not. The result of the empirical studies shows that our framework can assist the MFS identifying algorithm in getting a more accurate result.

The main contributions of this paper are:

1. We show that the fault corresponding to the MFS has different levels, i.e., fault with high level can mask the fault with low level.

2. We give a framework to assist algorithms to avoid the bad influence of the masking effect in different levels of fault.

3. We empirically studies that with considering the masking effect the algorithm can perform better than not considering the effect.

Rest of paper is organised as follows: section 2 gives a simple example to motivate our work. Section 3 describe our framework in detail. Section 4 illustrate the experiment and reports the result. Section 5 discusses the related works. Section 6 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;

  }
```

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

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 2: Identified MFSs and their corresponding Exception

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

Table 3: expected MFSs and their corresponding Exception

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

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.

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.

3.2 Dominate tree

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.

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

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 previous work

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.

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 Do not consider the masking effect

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.

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

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 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

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

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

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

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.

4.5 limitations in practice

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

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 decide 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:

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 ← t_original − s_fixed
3:     for each opt ∈ s_mutant do
4:         i = getIndex(Param, opt)
5:         opt ← opt' s.t. o ∈ Param[i] and opt'! = opt
6:     end for
7:     t_new ← s_fixed ⋃ s_mutant
8:     result ← 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
```

This algorithm take the a test configuration which trigger a unexpected fault, the fixed part, the fault type and the configuration paramters as input. The loop can been part into two parts:

first part(line 2 - line 7), it generate a new test configuration which 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 value that is different from the original one, and at the same time must take the value that is legal(line 3 - 6). The choose process is just by random and generated test configuration must be different each iteration(can be implemented by hashing method).

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

It is note that the loop have another end extra besides we have find a expected test configuration(line 10), that is the first line of the loop(line). That is the when function *MeetEndCriteria()* return a true value. We didn't explictly show what the function *MeetEndCriteria()*, This is consider the power of the computer, we can exhaustive run all the possible test configuraions, but it may depend on what your computer and what the problem like. In this paper, we just choose a number 3 to repeat this function. When it ended with this extra, we will return null(line 15), which means we cannot find a expected test configuration.

Table 9: Identifying MFS using traditional OFOT

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 0 1 0	Pass
0 0 0 1	Pass
Mfs identified	
(- 0 0 0) for Err 1	

Table 10: 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
Mfs identified	
(- - 0 0) for Err 1	

5.2 Examples when apply this approach into some MFS identifying algorithms

For example,

For OFOT:

For CTA, usually it is a covering array. IF we cannot change it with pass or expected fault, we just dicard them as they can't be the schemas for the current fault.

Table 11: cta example

id	test inputs	result
1	(1, 0, 0, 0)	PASS
2	(0, 1, 0, 1)	EX 1
3	(1, 0, 1, 1)	EX 2
4	(0, 1, 1, 0)	PASS

Table 12: cta aug

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

For FIC, as it don't generate first, instead it generate test configurations according to the last time running example. So we should generate test configuration after each it feed us a fixed part.

For example,

Traditional MFS identifying algorithms all including a step: fix some parts of the original failing test configuration and then mutant other parts to generate newly test configuration. This step is the basis of these algorithms, as it can help them to compare other configurations, to isolate the failure-inducing parts, to help rank the possible schemas and so on. Our approach is to improve the quality of generated test configuration, so that to assist these algorithm to reduce the influence of the faults masking effects.

As a overview of our approach, our approach is listed as figure 5.2:

This figure tell us when given a original failing test configuration and the fixed part, our approach will repeat generating test configurations which mutant the elements in the original test configuration except contain this fixed part until the end condition is matched.

1. Mutant other parts of the original test configurations

Table 13: Identifying MFS using FIC_BS 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
Mfs identified				
(-	-	0	0)	for Err 1

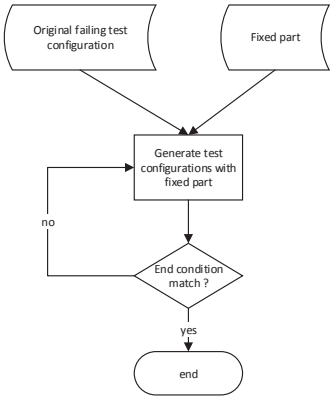

Figure 1: the overview of our approach.

2. Execute the SUT under the test configuration make some according to the result.

original failing test configuration. and fixed part.

generate test configurations.

judge the based on reuslt

end. regenerate.

In a word, our approach takes a revalidation strategy when encouter different faults until we find a test case. Next we will discuss the two parts in detail.

5.3 judge the result

There are three condition we will end our algorithm:

1. If the generated test configuration passed during testing. This means that the fixed part we validated does not contain the failure-inducing elements, so we will send a message to tell the MFS identifying algorithm that this fixed part is a healthy part, and return this passing generated test configuration to these algorithms.

2.If the generated test configuration failed with the same faulty information as the original test configuration.

3.We cannot generate anymore test configurations contained this fixed part and diff from the original test configuration.

In this case, we didn't know which condition does this fixed part should be because we didn't find any test configuration that contain this part and either passed or failed with the same fault information. In this paper, we default set this fixed part as the failure-inducing . And return any test configuration generated.

In fact, to be efficiency, we set the repeat times as a fixed number, say 3 to be end as soon as possible.

Reversely, the only condition that we did not stop is that we encounter a different fault information as the original test configuration, for in that case we did not has enough information whether this fixed part is a failure-inducing part or not as the different fault information may be a masking effect.

5.4 regenerate

The regenerate step is to regenerate test configuration that contain the fixed part and does not contain any same elements in the remained part. The reason is that if so we can't not make sure the fixed part or the contained remained part is the failure-inducing part.

Further more, as we having set a fixed number, we can't running all the possible test configuration, so if we did not encounter the passed or the same fault, we will take some more test configuration instead all of them to test. As similar test configuration will trigger the same fault, so we vary them as different as possible to avoid the following condition:

all the regenerated test configuration trigger a fault that have a higher level that the original one. The method we generate the test configuration it using random method. that is, using random number to take these unfixed part.

We generate as different as possible test cases for the tuple, to avoid some more MFSs that may mask this tuple. To do this we maintain a hash table to record the tuple and its corresponding test cases generated, each time we generate a random test case for this tuple, we will lookup this table to find if this test case is already generated, thus we will avoid generating redundancy test cases.

we will give simple example next to illustrate our approach. Take the fic as a example.

5.5 simple example

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 encouter a failure called "Err 1". Next we will take the MFS identifying algorithms – OFOT to identify the MFS for the "Err 1".

The process is listed in table 9. In this table, we can find the algorithm mutant one factor to take the different value from the original test configuration on time. Normally if the test configuration encounter the different condition with the Err 1, OFOT see 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 fouth factor, it doesn't trigger the Err 1 (for second factor, it trigger Err 2 and for the third and fourth, it passed). So finally we can get the MFS — (- 0 0 0).

Next, we will check how the ofot works with the assistance of our approach. The process is listed in table 10. We can soon find that the process is very similar to the original ofot algorithm, except that when we change the second factor to generate a new test configuration, we first encounter a new fault Err 2 which is different from the original Err 1, according to our approach, we will eliminate this test configuration and regenerate another one, when we find regenerated test configuration (0 2 0 0) failed with the same err – Err 1 with the original test configuration, we stopped the regenerate process for the fixed factor (0 - 0 0), and finally we send this test configuration (0 2 0 0) instead of (0 1 0 0) to the OFOT algorithm, and at last ofot get the more clear and precise MFS for Err 1 — (- - 0 0).

if the algorithm want to get the result, we will put off the result, and generate another one.

Five setup as follows:

Another note is that:

In detail, for FIC, TRT, OFOT, it can just use the original approach, that it each time test one tuple, and the we use our framework to assign this task.

For CTA, we can use our framework to choose a set of test cases for it to identify the MFS, like shykara do, but different from her work, we just not one test one time, we diff the test cases for different fault.

For SP, we can use our framework for him to generate test cases, it will test multiple tuples one time, and we just first for one fault, then another.

So our comparison work is just compare the one fault, with a noise for the multiple faults.

6. EMPIRICAL STUDIES

To evaluate our approach, we take two open-source software as our subjects: HSQLDB and Log4j. HSQLDB is a pure-java data-base management software, and Log4j is a logger tool commonly used in java application. Both of them have a large support developers' forum. Furthermore as our approach is a framework that can bee seen as a for the MFS identifying algorithms, so that we afford them five algorithms: fic ofot trt cta sp.

There are three questions we want to figure out from our in this empirical studies:

Q1: does our framework reduce the number of MFSs and the degree of MFS?

Q2: What the real masking state in real softwares.

Q3: does our framework the result is match the result in developer's forum.

6.1 experiment set up

We will first model the two objects so that we can use the MFS identifying algorithms. To do this, we first find real faults in the developer's forum, and then read the specification of the software to get the options and its values that can influence its behaviour. The detail of the modeling is listed in table 14 and table .

Second we will write the test script that can autonomically execute the software according to a given test configuration, furthermore, the test script will collect the executing result. In this paper, we just care two kinds of result: the test script run without any exception and run with triggering exception. For the second result we will distinguish the failure with different exception information.

Third we will take three group experiments, each for the question proposed in the first of this section.

For the first experiment, we will compare the result got by traditional MFS identifying algorithm with the MFS identifying algorithms using our framework. For a particular algorithm, say, OFOT, we will record the number of MFS, the degree of MFSs and the number of test configurations they needed.

The second experiment, we will collect the result got by the algorithms with our framework. Then we will generate a test configuration contain two MFS of the result, and execute to find if one MFS is masked by another. In specific, we will exam every possible pair MFSs, the only constraints is that this two pair MFS must get different fault information.

For the third experiment, we will compare the similarity between the identified result and the result report in the developer's forum. And find whose (traditional or with our framework) result can give more help to the developer of the software.

6.2 evaluation metrics

To evaluate the efficiency of our framework, we will compare the results of two approaches. We will count how many MFSs is reduced by our approach and how the degree is reduced by our approach compared to traditional one. This

Table 14: 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

two metric will indicate how much our approach will improve the usability of results. Additional, we will count how many more test configurations we will generated compare to traditional one, this metric indicate the cost we need over the traditional one.

Then we will exam the result of the algorithm with our framework, to see if any masking is behind them. In detail, we will record how many pair MFS can coexist in a test configuration. And count the privileges of each MFS.

For the third, the similarity is listed as follows: the followed notation: 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 $(-1\ 2\ -\ 3) \bigcap (-2\ 2\ -\ 3) = \{(-\ -\ 2\ -\ -), (-\ -\ -\ -\ 3)\}$.

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

.

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.

6.3 result and analysis

For the second experiment.

Table 15: Masking effect state

all pair	coexisted & noexised	mask over 6 mfs	over 5 mfs
10	5 & 5	3	2

7. RELATED WORKS

combinatorial testing has may factors, Nie give a survey, at some are focus,.

Identify MFS are

then consider the masking effect is , to my best knowledge, only one paper, unfortunately, it just consider the .

Ylimaz propose a work that is feedback driven combinatorial testing, different from our work, it first using CTA classify the possible MFS and then elimate them and generate new test cases to detect possible masked interaction in the next iteration. The difference 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.

8. CONCLUSIONS

This paragraph will end the body of this sample document. Remember that you might still have Acknowledgments or Appendices; brief samples of these follow. There is still the Bibliography to deal with; and we will make a disclaimer about that here: with the exception of the reference to the LaTeX book, the citations in this paper are to articles which have nothing to do with the present subject and are used as examples only.

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. ADDITIONAL AUTHORS

Additional authors: John Smith (The Thørväld Group, email: `jsmith@affiliation.org`) and Julius P. Kumquat (The Kumquat Consortium, email: `jpkumquat@consortium.net`).

11. REFERENCES

APPENDIX

A. HEADINGS IN APPENDICES

The rules about hierarchical headings discussed above for the body of the article are different in the appendices. In the **appendix** environment, the command **section** is used to

A.1 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references) to create the .bbl file. Insert that .bbl file into the .tex source file and comment out the command `\thebibliography`.

B. MORE HELP FOR THE HARDY

The sig-alternate.cls file itself is chock-full of succinct and helpful comments. If you consider yourself a moderately experienced to expert user of LaTeX, you may find reading it useful but please remember not to change it.