

Identify minimal failure-causing schemas for multiple faults

XINTAO NIU and CHANGHAI NIE, State Key Laboratory for Novel Software Technology, Nanjing University

HARETON LEUNG, Hong Kong Polytechnic University

JEFF Y. LEI, Department of Computer Science and Engineering, The University of Texas at Arlington

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). To extend and make full use of CT, the theory of Minimal Failure-Causing Schema(MFS) has been proposed. The use of MFS helps to isolate the root cause of the failure, which is desired after detecting them by CT. Most existed algorithms based on MFS theory focus on identifying the MFS in SUT with the single fault, however, we argue that multiple faults is the more common testing scenario, and under which masking effects may be triggered so that some expect faults will not be observed normally. Traditional MFS theory as well as its identifying algorithms lack a mechanism to handle such effects, hence will make them incorrectly isolate the MFS in SUT. To address this problem, we proposed a new MFS model with considering multiple faults. We first formally analyse the impact of the multiple faults on existed MFS isolating algorithms, especially when masking effects were triggered between these multiple faults. Based on this, we then give an approach that can assist traditional algorithms to better handle the multiple faults testing scenarios. Empirical studies with several open-source software were conducted, which show that multiple faults with masking effects do negatively affect on traditional MFS identifying approach and our approach can help them to alleviate these effects.

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

General Terms: Reliability, Verification

Additional Key Words and Phrases: Software Testing, Combinatorial Testing, Failure-causing schemas, Masking effects

ACM Reference Format:

Xintao Niu, Changhai Nie and Hareton Leung, 2014. Identify failure-causing schemas for multiple faults. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2010), 39 pages.
DOI: <http://dx.doi.org/10.1145/0000000.0000000>

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 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. In the worst case we need to examine every possible combination of these factors as each such combination can contain unique faults[Song et al. 2012]. While conducting such exhaustive testing is ideal and necessary in theory, it is impractical and not economical in consideration

This work was supported by the National Natural Science Foundation of China (No. 61272079), the Research Fund for the Doctoral Program of Higher Education of China (No.20130091110032), the Science Fund for Creative Research Groups of the National Natural Science Foundation of China(No. 61321491), and the Major Program of National Natural Science Foundation of China (No. 91318301)

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2010 ACM 1539-9087/2010/03-ART39 \$15.00

DOI: <http://dx.doi.org/10.1145/0000000.0000000>

Table I. MS word example

id	Highlight	Status bar	Bookmarks	Smart tags	Outcome
1	On	On	On	Off	PASS
2	On	Off	Off	On	PASS
3	Off	On	Off	Off	Fail
4	Off	Off	On	Off	PASS
5	Off	On	On	On	PASS

of the limited testing time and computing resource. One remedy for this problem is combinatorial testing, which systematically sample the interaction space and select a relatively small set 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., the *strength* of the interaction. Many works in CT aims to construct the smallest set of such efficient testing object [Cohen et al. 1997; Bryce et al. 2005; Cohen et al. 2003], which also called *covering array*.

Once failures are detected by covering array, it is desired to isolate the failure-inducing combinations in these failing test cases. This task is important in CT as it can facilitate the debugging efforts by reducing the code scope that needed to inspected [Ghandehari et al. 2012]. Only with the information stum from the covering array sometimes is far from clear to identify the location and magnitude of the failure-inducing combinations [Colbourn and McClary 2008]. Thus, deeper analysis needed to be conducted. Take an example [Bach and Schroeder 2004], Fig I present a 2-way covering array for testing the MS word application, in which we want to examine various combination of options ‘Highlight’, ‘Status Bar’, ‘Bookmarks’ and ‘Smart tags’ of MS word. Assume the third test case failed, then we can get that there are five 2-way suspicious combinations may be responsible for this failure: (Highlight: Off, Bookmarks: Off), (Highlight: Off, Smart tags: Off), (Status Bar: On, Bookmarks: Off), (Status Bar: On, Smart tags: Off), (Bookmarks: Off, Smart tags: Off). (Note that (Highlight: Off, Status Bar: On) excludes in this set as it appeared in the fifth passing test case). Without any more information, we cannot figure out which one or some in this suspicious set caused this failure. In fact, taking account of the higher-strength combination, e.g., (Highlight: Off, Status Bar: On, Smart tags: Off), the problem will grow more complicated.

To address this problem, prior work [Nie and Leung 2011a] specifically studied the properties of the minimal failure-causing schemas in SUT, based on which, a further diagnosis with generating additional test cases was applied and therefore can identify the MFS in the test case. Other works have been proposed to identify the MFS in SUT, which include approaches such as building tree model [Yilmaz et al. 2006], exploiting the methodology of minimal failure-causing schema [Nie and Leung 2011a], ranking suspicious classification ous combinations based on some rules [Ghandehari et al. 2012], using graphic-based deduction [Martínez et al. 2008] and so on. These approaches can be partitioned into two categories [Colbourn and McClary 2008]: *adaptive*—test cases are chosen based on the outcomes of the executed tests [Nie and Leung 2011a; Ghandehari et al. 2012; Niu et al. 2013; Zhang and Zhang 2011; Shakya et al. 2012; Wang et al. 2010; Li et al. 2012] or *nonadaptive*—test cases are chosen independently and can be executed parallel [Yilmaz et al. 2006; Colbourn and McClary 2008; Martínez et al. 2008; 2009; Fouché et al. 2009].

The MFS methodology as well as other MFS identifying approaches mainly focus on the ideal scenario that SUT only contains one fault, i.e., the test case under testing can either fails or passes the testing. However, in this paper, we argue that SUT with multiple distinguished faults is the more common testing scenario in practice, and moreover, this do have impact on the Failure-inducing Combinations Identifying (FCI) approaches. One main impact of multiple faults on FCI approaches is the masking ef-

fects. A masking effect [Dumlu et al. 2011; Yilmaz et al. 2013] is an effect that some failures prevents test cases from normally checking combinations that are supposed to be tested. Take the Linux command– *Grep* for example, we noticed that there are two different faults reported in the bug tracker system. The first one ¹ claims that *Grep* incorrectly match unicode patterns with ‘\<\>’, while the second one ² claims an incompatibility between option ‘-c’ and ‘-o’. When we put this two scenarios into one test case, only one fault information will be observed, which means another fault is masked by the observed one. This effects will prevent test cases normally executing, consequently make approaches make a incorrectly judgements on the correlation between the combinations checked in the test case and the fault that been prevented to be observed.

As known that masking effects negatively affect the performance of FCI approaches, a natural question is how this effect bias the results of FCI approaches. In this paper, we formalized the process of identifying MFS under the circumstance that masking effects exist in SUT and try to answer this question. One insight from the formal analysis 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 are harmful for FCI process.

Based on the insight we proposed a strategy to alleviate this impact. This strategy adopts the divide and conquer framework, i.e., separately handles each fault in SUT. For a particular fault under analysis, when applying traditional FCI approaches to identify the failure-inducing combinations, we pick the test cases generated by FCI approaches that trigger unexpected faults and replace them with regenerated newly test cases. These newly test cases should either pass or trigger the same fault under analysis.

The key to our approach is to search a test case that do not trigger unexpected faults that may import the masking effect. To guide the searching process, i.e., to reduce the possibility that the extra generated test case triggering unexpected fault, a natural idea is to learn some characteristics from the existed test cases and make the characteristic of the newly searched test case as much as different from those existed test cases which triggered unexpected fault. To reach this target, we define the *related strength* between the factor and the faults, for which the more the *related strength* is between a factor and a particular fault, the more the factor is likely to trigger this fault. We then using the integer linear programming(ILP) technique to find a test case which has the smallest *related strength* between unexpected fault.

To evaluate the performance of our approach, we applied our strategy on the FCI approach FIC_BS [Zhang and Zhang 2011]. The subjects we used are several open-source software with the developers’ forum in Source-Forge community. Through studying their bug reports in the bug tracker system as well as their user’s manual guide, we built the testing model which can reproduce the reported bugs with specific test cases. We then respectively compare the FCI approach augment with our strategy (AUGFCI) with traditional FCI approach with these subjects. We further empirically studied the performance of the important component of our strategy – searching satisfied test cases, to conduct this study, we compare our AUGFCI approach with the FCI approach augment by randomly searching satisfied test case. We finally compared our approach with the existed masking handling technique – FDA-CIT[Yilmaz et al. 2013]. All of these empirical studies showed that our replacing strategy as well as the searching test case component get a better performance than these traditional approaches when

¹<http://savannah.gnu.org/bugs/?29537>

²<http://savannah.gnu.org/bugs/?33080>

```

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

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

    return x+y;
}

```

Fig. 1. A toy program with four input parameters

the subject is suffering multiple faults, especially when these faults can import masking effects.

The main contributions of this paper are:

- We studied the impact of the masking effects among multiple faults on the isolation of the failure-inducing combinations in SUT.
- We proposed a divide and conquer strategy of selecting test cases to alleviate the impact of this effects.
- We designed an efficient test case searching method which can rapidly find a test case that does not trigger unexpected fault.
- We conducted several empirical studies and showed that our strategy can assist FCI approaches to get better performance on identifying failure-inducing combinations in SUT with masking effects.

2. MOTIVATING EXAMPLE

This section constructed a small program example, for convenience 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: $v_a = \{7, 11\}$, $v_b = \{2, 4, 5\}$, $v_c = \{4, 6\}$, $v_d = \{3, 5\}$. The detail code of the method is listed in Figure 1.

Inspecting the simple code in Figure 1, we can find two potential faults: First, in the step 1 we can get an *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 failure-inducing combinations in this example should be (-, 4, 4, -) and (-, -, 4, 5).

Traditional FCI algorithms do not consider the detail of the code, instead, they apply black-box testing to test 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 combinations for a particular fault must only appear in those inputs that trigger this fault. As traditional FCI approaches aim at using as few inputs as possible to get the same or approximate result as exhaustive testing, so the results derived from a exhaustive testing set must be the best that these FCI approaches can reach. Next we will illustrate how exhaustive testing works on identifying the failure-inducing combinations in the program.

We first generate every possible inputs listed in the Column “test inputs” of Table II, and their execution results are listed in Column “result” of Table II. In this Column, *PASS* means that the program runs without any exception under the input in the same row. *Ex 1* indicates that the program triggered an exception corresponding to the step 1 and *Ex 2* indicates the program triggered an exception corresponding to the

Table II. test inputs and their corresponding result

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

Table III. Identified failure-inducing combinations and their corresponding Exception

Failure-inducing combinations	Exception
(-, 4, 4, -)	Ex 1
(-, 2, 4, 5)	Ex 2
(-, 3, 4, 5)	Ex 2

step 2. According to data listed in table II, we can figure out that $(-, 4, 4, -)$ must be the failure-inducing combination of Ex 1 as all the inputs triggered Ex 1 contain this combination. Similarly, the combination $(-, 2, 4, 5)$ and $(-, 3, 4, 5)$ must be the failure-inducing combinations of the Ex 2. We listed this three combinations and their corresponding exception in Table III.

Note that we didn't get the expected result with traditional FCI approaches in this case. The failure-inducing combinations we get for Ex 2 are $(-, 2, 4, 5)$ and $(-, 3, 4, 5)$ respectively instead of the expected combination $(-, -, 4, 5)$. So why we failed in getting the $(-, -, 4, 5)$? The reason lies in *input 6*: $(7, 4, 4, 5)$ and *input 18*: $(11, 4, 4, 5)$. This two inputs contain the combination $(-, -, 4, 5)$, but didn't trigger the Ex 2, instead, Ex 1 was triggered.

Now let us get back to the source code of *foo*, we can find that if Ex 1 is triggered, it will stop executing the remaining code and report the exception information. In another word, Ex 1 has a higher fault level than Ex 2 so that Ex 1 may mask Ex 2. Let us re-examine the combination $(-, -, 4, 5)$: if we supposed that *input 6* and *input 18* should trigger Ex 2 if they didn't trigger Ex 1, then we can conclude that $(-, -, 4, 5)$ should be the failure-inducing combination of the Ex 2, which is identical to the expected one.

However, we cannot validate the supposition, i.e., *input 6* and *input 18* should trigger Ex 2 if they didn't trigger Ex 1, unless we have fixed the code that trigger Ex 1 and then re-executed all the test cases. So in practice, when we do not have enough resource to re-execute all the test cases again and again or can only take black-box testing, the more economic and efficient approach to alleviate the masking effect on FCI approaches is desired.

3. FORMAL MODEL

This section presents some definitions and propositions to give a formal model for the FCI problem.

3.1. Failure-causing Schemas in CT

Assume that the SUT is influenced by k parameters, and each parameter p_i has a_i discrete values from the finite set V_i , i.e., $a_i = |V_i|$ ($i = 1, 2, \dots, k$). Some of the definitions below are originally defined in [Nie and Leung 2011b].

Definition 3.1. A *test case* of the SUT is an array of k values, one for each parameter of the SUT, which is denoted as a k -tuple (v_1, v_2, \dots, v_k) , where $v_1 \in V_1, v_2 \in V_2 \dots v_k \in V_k$.

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

We consider the fact that the abnormally executing test cases as a *fault*. It can be a thrown exception, compilation error, assertion failure or constraint violation. When faults are triggered by some test cases, what is desired is to figure out the cause of these faults, and hence some subsets of this test case should be analysed.

Definition 3.2. For the SUT, the t -tuple $(-, v_{k_1}, \dots, v_{k_t}, \dots)$ is called a t -value *schema* ($0 < t \leq k$) when some t parameters have fixed values and the others can take on their respective allowable values, represented as “-”.

In effect a test case itself is a t -value *schema*, when $t = k$. Furthermore, if a test case contain a *schema*, i.e., every fixed value in the combination is in this test case, we say this test case *hits* the *schema*.

Definition 3.3. let c_l be a l -value schema, c_m be an m -value schema in SUT and $l < m$. If all the fixed parameter values in c_l are also in c_m , then c_m *subsumes* c_l . In this case we can also say that c_l is a *sub-schema* of c_m and c_m is a *parent-schema* of c_l , which can be denoted as $c_l \prec c_m$.

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

Definition 3.4. If all test cases contain a schema, say c , trigger a particular fault, say F , then we call this schema c the *failure-causing schema* for F . Additionally, if none sub-schema of c is the *failure-causing schema* for F , we then call the schema c the *Minimal Failure-causing Schema*, i.e., (MFS) for F .

In fact, MFS is identical to the failure-inducing combinations we discussed previously. Figuring it out can eliminate all details that are irrelevant for causing the failure and hence facilitate the debugging efforts.

Some notions will be used later are listed in the following for the convenience of reference:

- k : the number of parameters that influence the SUT.
- V_i : the set of discrete values that the i th factor of the SUT can take.
- T^* : The exhaustive set of test cases for the SUT. For a SUT with k factors, and each factor can take $|V_i|$ values, the number of this set of test cases T^* is $\prod_{i=1}^{i=k} |V_i|$.
- L : the number of faults that contain in the SUT.
- F_m : the m th fault in the SUT, for different faults, we can differentiate them from their exception traces or other buggy information.
- T_{F_m} : All the test cases that can trigger the fault F_m .
- $\mathcal{T}(c)$: All the test cases that can hit(contain) the schema c . Based on the definition of MFS, we can learn that if schema c is MFS for F_m , then $\mathcal{T}(c)$ must be subsumed in T_{F_m} .
- $\mathcal{I}(t)$: All the schemas that are hit in the test case t , e.g., $\mathcal{I}((111)) = \{(1-)(-1-)(--1)(11-)(1-1)(-11)(111)\}$.
- $\mathcal{I}(T)$: All the schemas that are hit in a set of test cases T , i.e., $\mathcal{I}(T) = \bigcup_{t \in T} \mathcal{I}(t)$.
- $\mathcal{S}(T)$: All the schemas that are only hit in the set of test cases, it is important to note that this set is different from $\mathcal{I}(T)$, as the schemas hit by the test cases in T can also

Table IV. Example of the proposition 3.5

c	
$(0, 0, -, -)$	$\mathcal{T}(c)$
$(0, 0, 0, 0)$	$(0, 0, 0, 0)$
$(0, 0, 0, 1)$	$(0, 0, 0, 1)$
$(0, 0, 1, 0)$	$(0, 0, 1, 0)$
$(0, 0, 1, 1)$	$(0, 0, 1, 1)$
$(0, 1, 0, 0)$	$(0, 1, 0, 0)$
$(0, 1, 0, 1)$	$(0, 1, 0, 1)$
$(0, 1, 1, 0)$	$(0, 1, 1, 0)$
$(0, 1, 1, 1)$	$(0, 1, 1, 1)$

be hit by other test cases that are not belong to this set. In fact, $\mathcal{S}(T)$ is computed by $\{c | c \in \mathcal{I}(T) \text{ and } c \notin \mathcal{I}(T^* \setminus T)\}$.

— $\mathcal{C}(T)$: the minimal schemas only hit by the set of test cases T , this set is the sub-set of $\mathcal{S}(T)$, which is defined as $\{c | c \in \mathcal{S}(T) \text{ and } \nexists c' \prec c, s.t., c' \in \mathcal{S}(T)\}$.

PROPOSITION 3.5. *For l -value schema c_l and m -value schema c_m , if $c_l \prec c_m$, then we can have all the test cases hit c_m must also hit c_l , i.e., $\mathcal{T}(c_m) \subset \mathcal{T}(c_l)$.*

PROOF. Suppose $\forall t \in \mathcal{T}(c_m)$, we have that t hits c_m . Then as $c_l \prec c_m$, it must have t must also hit c_l . This is because all the elements in c_l are also in c_m , which are contained in the test case t . Therefore we get $t \in \mathcal{T}(c_l)$. Thus $t \in \mathcal{T}(c_m)$ implies $t \in \mathcal{T}(c_l)$, so it follows that $\mathcal{T}(c_m) \subset \mathcal{T}(c_l)$. \square

Table IV illustrates an example of SUT with four binary parameters (Unless otherwise specified, following examples also assume the SUT with binary parameters). The left column lists the schema $(0,0,-,-)$ as well as all the test cases hit this schema, while the right column lists the test cases for schema $(0,-,-,-)$. We can observe that $(0,-,-,-) \prec (0,0,-,-)$ and the the set of test cases which hit $(0,-,-,-)$ contains the set that hits $(0,0,-,-)$.

PROPOSITION 3.6.

For any set T of test cases of a SUT, we can always get a set of minimal schemas $\mathcal{C}(T) = \{c | \nexists c' \in \mathcal{C}(T), s.t. c' \prec c\}$, such that,

$$T = \bigcup_{c \in \mathcal{C}(T)} \mathcal{T}(c)$$

PROOF. We prove by producing this set of schemas.

We have denoted the exhaustive test cases for SUT as T^* and let $T^* \setminus T$ be the test cases that in T^* but not in T . It is obviously $\forall t \in T$, we can always find at least one schema which hit by t , i.e., $c \in \mathcal{I}(t)$, such that $c \notin \mathcal{I}(T^* \setminus T)$. Specifically, at least the test case t itself as schema holds.

Then we collect all the satisfied schemas which only hit by the test cases in the test cases of T , which can be denoted as: $\mathcal{S}(T) = \{c | c \in \mathcal{I}(T) \text{ and } c \notin \mathcal{I}(T^* \setminus T)\}$.

For the schemas in $\mathcal{S}(T)$, we can have $\bigcup_{c \in \mathcal{S}(T)} \mathcal{T}(c) = T$. This is because first, for $\forall t \in \mathcal{T}(c), c \in \mathcal{S}(T)$, it must have $t \in T$. This is because if not so, then $t \in T^* \setminus T$, which contradict with the definition of $\mathcal{S}(T)$. So $t \in T$. Hence, $\bigcup_{c \in \mathcal{S}(T)} \mathcal{T}(c) \subset T$.

Then second, for any test case t in T , as we have learned at least find one c' in $\mathcal{I}(t)$, such that $c' \in \mathcal{S}(T)$ (The t itself as a schema holds). In another word, the test case t hit the schema c' , which implies $t \in \mathcal{T}(c'), c' \in \mathcal{S}(T)$. And obviously $\mathcal{T}(c') \subset \bigcup_{c \in \mathcal{S}(T)} \mathcal{T}(c)$, so $t \in \bigcup_{c \in \mathcal{S}(T)} \mathcal{T}(c)$, therefore, $T \subset \bigcup_{c \in \mathcal{S}(T)} \mathcal{T}(c)$.

Since $\bigcup_{c \in \mathcal{S}(T)} \mathcal{T}(c) \subset T$ and $T \subset \bigcup_{c \in \mathcal{S}(T)} \mathcal{T}(c)$, so it follows $\bigcup_{c \in \mathcal{S}(T)} \mathcal{T}(c) = T$.

Table V. Example of the minimal schemas

T	$S(T)$	$\mathcal{C}(T)$
(0, 0, 0, 0)	(0, 0, 0, 0)	(0, 0, 0, -)
(0, 0, 0, 1)	(0, 0, 0, 1)	(0, 0, -, 0)
(0, 0, 1, 0)	(0, 0, 1, 0)	
	(0, 0, 0, -)	
	(0, 0, -, 0)	

Then we denote the minimal schemas of $S(T)$ as $M(S(T)) = \{c | c \in S(T) \text{ and } \nexists c' \prec c, s.t., c' \in S(T)\}$. For this set, we can still have $\bigcup_{c \in M(S(T))} \mathcal{T}(c) = T$. We also prove this by two steps, first and obviously, $\bigcup_{c \in M(S(T))} \mathcal{T}(c) \subset \bigcup_{c \in S(T)} \mathcal{T}(c)$. Then we just need to prove that $\bigcup_{c \in S(T)} \mathcal{T}(c) \subset \bigcup_{c \in M(S(T))} \mathcal{T}(c)$.

In fact by definition of $M(S(T))$, for $\forall c' \in S(T) \setminus M(S(T))$, we can have some $c \in M(S(T))$, such that $c \prec c'$. According to the Proposition 3.5, $\mathcal{T}(c') \subset \mathcal{T}(c)$. So for any test case $t \in \bigcup_{c \in S(T)} \mathcal{T}(c)$, as we have either $\exists c' \in S(T) \setminus M(S(T)), s.t., t \in \mathcal{T}(c')$ or $\exists c \in M(S(T)), s.t., t \in \mathcal{T}(c)$. Both cases can deduce $t \in \bigcup_{c \in M(S(T))} \mathcal{T}(c)$. So, $\bigcup_{c \in S(T)} \mathcal{T}(c) \subset \bigcup_{c \in M(S(T))} \mathcal{T}(c)$.

Hence, $\bigcup_{c \in S(T)} \mathcal{T}(c) = \bigcup_{c \in M(S(T))} \mathcal{T}(c)$, and $M(S(T))$ is the set of schemas that holds this proposition. \square

For example, table V lists the $S(T)$ and minimal schemas $\mathcal{C}(T)$ for the set of test cases T . We can see that for any other schemas not in $\mathcal{C}(T)$, either can we find a test case not in T hit the schema, e.g., (0,0,-,-) with the test case (0,0,1,1) not in T , or is the parent schema of one of this two minimal schemas, e.g., (0,0,0,0) the parent schema of both (0,0,0,-) and (0,0,-,0).

Let T_{F_m} denotes the set of all the test cases triggering fault F_m , then $\mathcal{C}(T_{F_m})$ actually is the set of MFS of F_m by definition of MFS.

From the construction process of $\mathcal{C}(T)$, one observation is that the minimal schemas $\mathcal{C}(T)$ is the subset of the the schemas set $S(T)$, i.e., $\mathcal{C}(T) \subset S(T)$, and for any schema in $S(T)$, it either belongs to $\mathcal{C}(T)$, or is the parent schema of one element of $\mathcal{C}(T)$. Then we can have the following proposition.

PROPOSITION 3.7. *For any test cases set T and schema c , if any test case hit c is in the set T , i.e., $\mathcal{T}(c) \subset T$, then it must be that $c \in S(T)$.*

PROOF. We first have $c \in \mathcal{C}(\mathcal{T}(c))$, this is obviously and in fact the minimal schemas for the test cases set $\mathcal{T}(c)$ only contain one schema, which is exactly c itself. As discussed previously we have $\mathcal{C}(\mathcal{T}(c)) \subset S(\mathcal{T}(c))$, so it must be $c \in S(\mathcal{T}(c))$.

Then as $\mathcal{T}(c) \subset T$, it follows $S(\mathcal{T}(c)) \subset S(T)$ by definition. In detail, $S(\mathcal{T}(c)) = \{c | c \in \mathcal{I}(\mathcal{T}(c)) \text{ and } c \notin \mathcal{I}(T^* \setminus \mathcal{T}(c))\}$, so $S(\mathcal{T}(c)) \subset \{c | c \in \mathcal{I}(T) \text{ and } c \notin \mathcal{I}(T^* \setminus T)\}$, which is exactly $S(T)$.

So as $c \in S(\mathcal{T}(c))$ and hence $c \in S(T)$. \square

For two different set of test cases, there exist some relationships between the minimal schemas of these two set, which varies in the relevancy between this two different set of test cases. In fact, there are three possible associations between two different set of test cases: *inclusion*, *disjoint*, and *intersection* as listed in figure 2. We didn't list the condition that two set that are identical, because on that condition the minimal schemas must also be identical. To discuss the properties of the relationship of the minimal schemas between two different set of test cases is important as we can learn later the masking effects between multiple faults will make the MFS identifying process incorrectly works, i.e., these FCI approaches may isolate the minimal schemas for the set of test cases which are biased from the expected failing set of test cases. And

Table VI. Example of the scenarios

T_l	T_k	T_l	T_k
(0, 0, 0)	(0, 0, 0)	(0, 0, 0)	(0, 0, 0)
(0, 0, 1)	(0, 0, 1)	(0, 0, 1)	(0, 0, 1)
(0, 1, 0)	(0, 1, 0)	(0, 1, 0)	(0, 1, 0)
	(0, 1, 1)		(1, 1, 0)
			(1, 1, 1)
$\mathcal{C}(T_l)$	$\mathcal{C}(T_k)$	$\mathcal{C}(T_l)$	$\mathcal{C}(T_k)$
(0, 0, -)	(0, -, -)	(0, 0, -)	(0, 0, -)
(0, -, 0)		(0, -, 0)	(0, -, 0)
			(1, 1, -)

these properties can help us to figure out the impact of masking effects on the FCI approaches. Next we will separately discuss the relationship between minimal schemas under the three conditions.

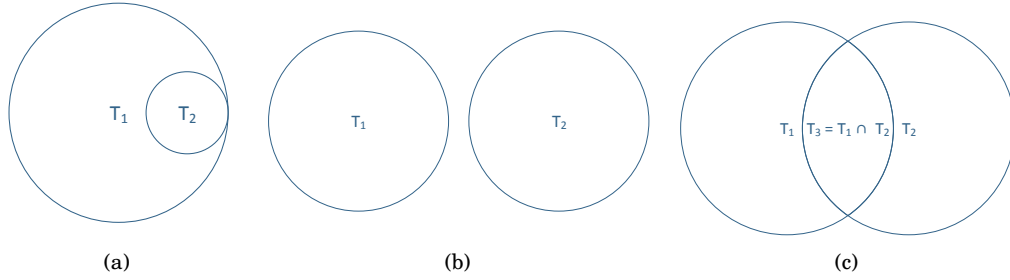


Fig. 2. Test Suites relationship

3.2. Inclusion

It is the first relationship corresponding to Fig. 2(a). We can have the following proposition with two set of test cases which have the inclusion relationship.

PROPOSITION 3.8. *For two set of test cases T_l and T_k , assume that $T_l \subset T_k$. Then we have*

$$\forall c_l \in \mathcal{C}(T_l) \text{ either } c_l \in \mathcal{C}(T_k) \text{ or } \exists c_k \in \mathcal{C}(T_k), \text{ s.t., } c_k \prec c_l.$$

PROOF. Obviously for $\forall c_l \in \mathcal{C}(T_l)$ we can get $\mathcal{T}(c_l) \subset T_l \subset T_k$. According to the proposition 3.7, we can have $c_l \in \mathcal{S}(T_k)$. So this proposition holds as the schema in $\mathcal{S}(T_k)$ either is also in $\mathcal{C}(T_k)$, or must be the parent of some schemas in $\mathcal{C}(T_k)$. \square

Based on this proposition, in fact, the schema $c_k \in \mathcal{C}(T_k)$ remains the following three possible relationships with $\mathcal{C}(T_l)$: 1) $c_k \in \mathcal{C}(T_l)$, or 2) $\exists c_l \in \mathcal{C}(T_l), \text{ s.t., } c_k \prec c_l$, or 3) $\nexists c_l \in \mathcal{C}(T_l), \text{ s.t., } c_k \prec c_l \text{ or } c_k = c_l, \text{ or } c_l \prec c_k$. For the third case, we call c_k is *irrelevant* to $\mathcal{C}(T_l)$.

We illustrate these scenarios in Table VI. There are two parts in this table, each part shows two set of test cases: T_l and T_k , which have $T_l \subset T_k$. For the left part, we can see that the schema in $\mathcal{C}(T_l)$: (0, 0, -) and (0, -, 0), both are the parent of the schema in $\mathcal{C}(T_k)$: (0, -, -). While for the right part, the schemas in $\mathcal{C}(T_l)$: (0, 0, -) and (0, -, 0) are both also in $\mathcal{C}(T_k)$. Furthermore, one schema in $\mathcal{C}(T_k)$: (1, 1, -) is irrelevant to $\mathcal{C}(T_l)$.

Table VII. Example of disjoint examples

T_l	T_k
(0, 0, 0)	(1, 0, 0)
(0, 1, 0)	(1, 0, 1)
	(1, 1, 0)
$\mathcal{C}(T_l)$	$\mathcal{C}(T_k)$
(0, -, 0)	(1, 0, -)
	(1, -, 0)

3.3. Disjoint

This relationship is corresponding to the Fig.2(b), For two different set of test cases, one obvious properties is listed as follows:

PROPOSITION 3.9. *For two test cases set T_1, T_2 , if $T_1 \cap T_2 = \emptyset$, we have, $\mathcal{S}(T_1) \cap \mathcal{S}(T_2) = \emptyset$.*

PROOF. If $\mathcal{S}(T_1) \cap \mathcal{S}(T_2) \neq \emptyset$. Without loss of generality, we let $c \in \mathcal{S}(T_1) \cap \mathcal{S}(T_2)$ we can learn that $\mathcal{T}(c)$ must both in T_1 and T_2 , which is contradiction. \square

This properties tells that the minimal schemas of two disjoint test cases should be irrelevant to each other, Table VII list the example of this scenario, we can learn in this table that for two different test cases set T_l, T_k , their minimal schemas, i.e., (0, -, 0) and (1, 0, -), (1, -, 0) respectively, are irrelevant to each other.

3.4. Intersect

This relationship is corresponding to the Fig.2(c). This scenario is the most common scenario for two set of test cases, but also the most complicated scenario for analysis. For convenience to illustrate the properties of the minimal schemas of this scenario, we assume that $T_1 \cap T_2 = T_3$ as depicted in Fig.2(c). Then we can have the following properties:

PROPOSITION 3.10. *For two intersecting set of test cases T_1 and T_2 (this two set is neither identical nor subsuming each other), it must have $\exists c_1 \in \mathcal{C}(T_1)$ and $c_2 \in \mathcal{C}(T_2)$. s.t. c_1 and c_2 are irrelevant.*

PROOF. Firstly, we can learn that $\mathcal{C}(T_1 \setminus T_3)$ are irrelevant to $\mathcal{C}(T_2 \setminus T_3)$, as $(T_1 \setminus T_3) \cap (T_2 \setminus T_3) = \emptyset$.

As $\mathcal{C}(T_1 \setminus T_3)$ is either identical to some schemas in $\mathcal{C}(T_1)$ or be parent schemas of them, then if some of them are identical, i.e., $\exists c', s.t., c' \in \mathcal{C}(T_1 \setminus T_3)$ and $c' \in \mathcal{C}(T_1)$, then these schemas c' must be irrelevant to $\mathcal{C}(T_2)$ as $(T_1 \setminus T_3) \cap T_2 = \emptyset$. This is also holds if $\mathcal{C}(T_2 \setminus T_3)$ is identical to some schemas in $\mathcal{C}(T_2)$.

Next, if both $\mathcal{C}(T_1 \setminus T_3)$ and $\mathcal{C}(T_2 \setminus T_3)$ are parent schemas of some of $\mathcal{C}(T_1)$ and $\mathcal{C}(T_2)$ respectively, without loss of generality, we let $c_1 \prec c_{1-3}$, ($c_{1-3} \in \mathcal{C}(T_1 \setminus T_3)$ and $c_1 \in \mathcal{C}(T_1)$) and $c_2 \prec c_{2-3}$, ($c_{2-3} \in \mathcal{C}(T_2 \setminus T_3)$ and $c_2 \in \mathcal{C}(T_2)$). Then these corresponding sub-schemas in $\mathcal{C}(T_1)$ and $\mathcal{C}(T_2)$, i.e., c_1 and c_2 respectively, must also be irrelevant to each other. This is because $\mathcal{T}(c_1) \supset \mathcal{T}(c_{1-3})$ and $\mathcal{T}(c_2) \supset \mathcal{T}(c_{2-3})$. And as $\mathcal{T}(c_{1-3}) \cap \mathcal{T}(c_{2-3}) = \emptyset$, so $\mathcal{T}(c_1)$ and $\mathcal{T}(c_2)$ are neither identical nor subsuming each other, which also implies that c_1 and c_2 is irrelevant to each other. \square

For example, Table VIII shows two test cases interact each other at test case (1,0,0), but their minimal schemas, (1,0,-) and (1,-,0) respectively, are irrelevant to each other.

PROPOSITION 3.11. *For two intersecting set of test cases T_1 and T_2 , and let $T_3 = T_1 \cap T_2$, if we can find $\exists c_1 \in \mathcal{C}(T_1)$ and $c_2 \in \mathcal{C}(T_2)$, s.t., c_1 is identical to c_2 , Then it must have $c_1 = c_2 \in \mathcal{C}(T_3)$*

Table VIII. Example of Intersect by irrelevant examples

T_1	T_2
(1, 0, 0)	(1, 0, 0)
(1, 0, 1)	(1, 1, 0)
$\mathcal{C}(T_1)$	$\mathcal{C}(T_2)$
(1, 0, -)	(1, -, 0)

Table IX. Example of Intersect by identical examples

T_1	T_2	$T_3 = T_1 \cap T_2$
(0, 1, 0)	(0, 0, 0)	(1, 1, 0)
(1, 1, 0)	(0, 0, 1)	(1, 1, 1)
(1, 1, 1)	(1, 1, 0)	
	(1, 1, 1)	
$\mathcal{C}(T_1)$	$\mathcal{C}(T_2)$	$\mathcal{C}(T_3)$
(-, 1, 0)	(0, 0, -)	(1, 1, -)
(1, 1, -)	(1, 1, -)	

PROOF. As we get that the identical schema must share the identical test cases, then the only identical test cases between T_1 and T_2 is $T_1 \cap T_2 = T_3$. So the only possible identical schemas between $\mathcal{C}(T_1)$ and $\mathcal{C}(T_2)$ is in $\mathcal{C}(T_3)$. \square

We must know that this proposition holds when some schemas in $\mathcal{C}(T_1 \cap T_2)$ is identical some schemas in $\mathcal{C}(T_1)$ and $\mathcal{C}(T_2)$.

For example, Table IX shows two test cases interact each other at test cases (1,1,0) and (1,1,1), and they have identical minimal schemas, i.e., (1,1,-), which is also the minimal schema in $\mathcal{C}(T_3)$.

PROPOSITION 3.12. *For two intersecting set of test cases T_1 and T_2 , and let $T_3 = T_1 \cap T_2$, if there we can find $\exists c_1 \in \mathcal{C}(T_1)$ and $c_2 \in \mathcal{C}(T_2)$, s.t., c_1 is parent-schema of c_2 , then it must have $c_1 \in \mathcal{C}(T_3)$. (and vice versa).*

PROOF. We have proved previously if two schemas have subsumes relationship, then their test cases must also have inclusion relationship. And as the only inclusion relationship between T_1 and T_2 is that $T_3 \subset T_1$ and $T_3 \subset T_2$. So the parent schemas must be in $\mathcal{C}(T_3)$. \square

It is noted that this proposition holds when holds when some schema in $\mathcal{C}(T_3)$ is identical in $\mathcal{C}(T_1)$ (or $\mathcal{C}(T_2)$), and simultaneously the same schemas in $\mathcal{C}(T_3)$ must be the parent-schema of the minimal schemas of another set of test cases, i.e., $\mathcal{C}(T_2)$ (or $\mathcal{C}(T_1)$).

Table X illustrates this scenario, in which, the minimal schemas of T_1 : (1,0,-), (1,-,0), which is also the schemas in $\mathcal{C}(T_3)$, is the parent schema of the minimal schema of T_2 : (1,-,-).

It is noted that this three condition can simultaneously appears when two set of test cases intersect with each other.

3.5. Identify the MFS

According to these analysis, we can get that $\mathcal{C}(T_{F_m})$ actually is the set of failure-causing schemas of F_m . Then in theory, if we want to accurately figure out the MFS in SUT, we need to exhaustively execute each possible test case, and collect the failing test cases T_{F_m} . This is impossible in practice especially when the testing space is too large.

So for traditional FCI approaches, they need to select the subset of the exhaustive test cases, and then either take some assumption to make prediction of the remaining

Table X. Example of Intersect by subsume examples

T_1	T_2	$T_3 = T_1 \cap T_2$
(0, 1, 0)	(0, 0, 0)	(1, 0, 0)
(1, 0, 0)	(1, 0, 0)	(1, 0, 1)
(1, 0, 1)	(1, 0, 1)	(1, 1, 0)
(1, 1, 0)	(1, 1, 0)	
	(1, 1, 1)	
$\mathcal{C}(T_1)$	$\mathcal{C}(T_2)$	$\mathcal{C}(T_3)$
(-, 1, 0)	(-, 0, 0)	(1, 0, -)
(1, 0, -)	(1, -, -)	(1, -, 0)
(1, -, 0)		

test cases or just give a suspicious rank. As giving a suspicious rank can also be regarded as a special case of making prediction (with computing the possibility), so we next only formally describe the mechanism of FCI approaches belong to the first type. We refer the observed failing test case to $T_{fail_{observed}}$, and refer the remaining test cases the approach predict to be failed as $T_{fail_{predicted}}$. We also denote the actually entire failing test cases as T_{fail} . Then the MFS identified by FCI approaches can be depicted as:

$$MFS = \mathcal{C}(T_{fail_{observed}} \cup T_{fail_{predicted}}).$$

For each FCI approach, the way it predict the $T_{fail_{predicted}}$ according to observed failing test cases varies, further more, as the test cases it generates is different, the failing test cases observed by different test cases, i.e., $T_{fail_{observed}}$ also varies. We take an example using OFOT approach to illustrate this formula.

Suppose SUT has 3 parameters, each can take 2 values. And assume the test case (1, 1, 1) failed. Then we can describe the FCI process as Table XI. In this table, test case t failed, and OFOT mutated one factor of the t one time to generate new test cases: t_1, t_2, t_3 . It found the t_1 passed, which indicates that this test case break the MFS in the original test case t . So the (1,-,-) should be one failure-causing factor, and as other mutating process all failed, which means no other failure-inducing factors were broken, therefore, the MFS in t is (1,-,-).

Now let us explain this process with our formal model. Obviously the $T_{fail_{observed}}$ is $\{(1,1,1), (1,0,1), (1,1,0)\}$. And as having found (0,-,-) broke the MFS, hence by theory [Nie and Leung 2011a], all the test cases contain (0,-,-) should pass the test cases (This conclusion is built on the assumption that SUT just contain one failure-causing schema). As a result, (0,1,1), (0,0,1), (0,1,0), (0,0,0) should pass the testing. Further, as obviously the test case either passes or fails the testing (we label the skip the testing as a special case of failing), so the remaining test case (1,0,0), will be predicted as failing, i.e., $T_{fail_{predicted}}$ is $\{(1,0,0)\}$. Taking together, the MFS using OFOT strategy can be described as: $\mathcal{C}(T_{fail_{observed}} \cup T_{fail_{predicted}}) = \mathcal{C}(\{(1, 1, 1), (1, 0, 1), (1, 1, 0), (1, 0, 0)\}) = (1, -, -)$, which is identical to the one it got previously.

Table XI. OFOT with our strategy

original test case				Outcome
t	1	1	1	Fail
observed				
t_1	0	1	1	Pass
t_2	1	0	1	Fail
t_3	1	1	0	Fail
predicted				
t_4	0	0	1	Pass
t_5	0	1	0	Pass
t_6	1	0	0	Fail
t_7	0	0	0	Pass

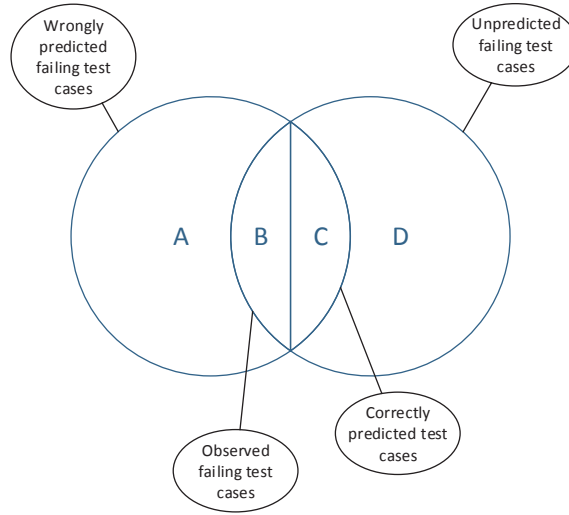


Fig. 3. generally model of FCI

Similarly, other FCI approaches can also be modeled into this formal description. We will not in detail discuss how to model each FCI approach as this is not the point of this paper. It is noted that the test cases FCI predict to be failing is not always identical to the actually failing test cases. In fact, we can generally depict the process of FCI approaches like Fig. 3.

We can see in Fig. 3, area A denotes the test cases that should be passed test cases while it predicted to be failed ones, area B depicts the test cases that the approach observed to be failed test cases, area C refers to the failed test cases that doesn't to be observed but be predicted to be failed test cases and area D illustrates the failed test cases that neither to be observed nor to be predicted. This figure is actually one sample of the condition that two set of test cases intersect with each other, in specific, area $A \cup B \cup C = T_1$, area $D \cup B \cup C = T_2$ and area $B \cup C = T_1 \cap T_2 = T_3$.

We learned previously this scenario makes the schemas identified in T_1 biased from the expected MFS in T_2 , in specific, their must be irrelevant schemas between $C(T_1)$ and $C(T_2)$, which means that FCI approach will identify some minimal schemas that is irrelevant to actual MFS, and must ignore some actual MFS. Moreover, under the appropriate condition list in proposition 3.11 and 3.12, FCI may identify the identical schemas or parent-schema or sub-schema of the actual MFS. So in order to identify the schemas as accurate as possible, the FCI approach needs to make T_1 as similar as possible to T_2 , specifically, makes area B and area C as large as possible, and make area A as small as possible.

However, even though each FCI approach tries its best to identify the MFS as accurate as possible, masking effects raised from test cases will make their efforts in vain. We next will discusses the masking problem and how it affect the FCI approaches.

Table XII. masking effects for exhaustive testing

T_1	$T_{mask(1)}$	T_*
(1, 1, 1, 1)	(1, 1, 0, 0)	(0, 1, 0, 0)
(1, 1, 1, 0)	(0, 1, 1, 1)	(0, 0, 0, 0)
(1, 1, 0, 1)		(1, 0, 0, 0)
		(1, 0, 1, 1)
		(0, 0, 1, 1)
actual MFS for 1 $\mathcal{C}(T_1 \cup T_{mask(1)})$	regard as one fault $\mathcal{C}(T_1 \cup T_{mask(1)} \cup T_*)$	distinguish faults $\mathcal{C}(T_1)$
(1, 1, -, -)	(-, -, 0, 0)	(1, 1, -, 1)
(-, 1, 1, 1)	(1, 1, -, -)	(1, 1, 1, -)
	(-, -, 1, 1)	

4. MASKING EFFECT

Definition 4.1. A *masking effect* is the effect that while a test case t hit a MFS for a particular fault, however, t didn't trigger the expected fault because other fault was triggered ahead which prevents t to be normally checked.

Taking the masking effects into account, when identifying the MFS for a specific fault, say, F_m , we should not ignore these test cases which should have triggered F_m if they didn't trigger other faults. We call these test cases $T_{mask(F_m)}$. Hence, the MFS for fault F_m should be $\mathcal{C}(T_{F_m} \cup T_{mask(F_m)})$.

As an example, in the motivation example in section 2, the $F_{mask(F_{Ex2})}$ is $\{(7,4,4,5), (11,4,4,5)\}$. So the MFS for $Ex2$ is $\mathcal{C}(T_{F_{Ex2}} \cup T_{mask(F_{Ex2})})$, which is $(-, -, 4, 5)$.

In practice with masking effects, however, it is not possible to correctly identifying the MFS, unless we fix some bugs in the SUT and re-execute the test cases to figure out $T_{mask(F_m)}$.

In effect for traditional FCI approaches, without the knowledge of $T_{mask(F_m)}$, only two strategies can be adopted when facing the multiple faults problem. We will separately analyse the two strategies under exhaustive testing condition and normal FCI testing condition.

4.1. masking effects for exhaustive testing

4.1.1. Regard as one fault. The first one is the most common strategy, it doesn't distinguish the faults, i.e., regard all the types of faults as one fault-*failure*, others as *pass*.

With this strategy, The minimal schemas we identify are the set $\mathcal{C}(\bigcup_{i=1}^L T_{F_i})$, L is the number of all the faults in the SUT. Obviously, $T_{F_m} \cup T_{mask(F_m)} \subset \bigcup_{i=1}^L T_{F_i}$. So in this case, by Proposition 3.8, some schemas we get may be the sub-schemas of some of the actual MFS, or irrelevant to the actual MFS.

As an example, consider the test cases in Table XII. In this example, assume we need to characterize the MFS for error 1, all the test cases that triggered error 1 is listed in the column T_1 , similarly, we list the test cases that triggered other fault in column $T_{mask(1)}$ and T_* respectively, in which the former masked the error 1 while the latter not. Actually the MFS for error 1 should be $(1,1,-,-)$ and $(-,1,1,1)$ as we listed them in the column *actual MFS for 1*. However, when we take *regard as one fault* strategy, the minimal schemas we got will be $(-, -, 0, 0)$, $(1,1,-,-)$, $(-, -, 1, 1)$, in which the $(-, -, 0, 0)$ is irrelevant to the actual MFS for error 1 and the $(-, -, 1, 1)$ is the sub-schema of the actual MFS $(-, 1, 1, 1)$.

4.1.2. Distinguish faults. Distinguishing the faults by the exception traces or error code can help make the MFS related to particular fault. Yilmaz in [Yilmaz et al. 2013] proposed the *multiple-class* failure characterizing method instead of *ternary-class* ap-

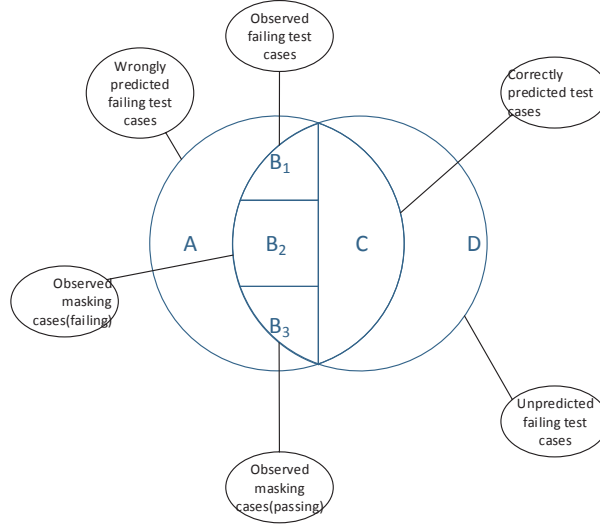


Fig. 4. FCI with masking effects

proach to make the characterizing process more accurately. Besides, other approaches can also be easily extended with this strategy to be applied on SUT with multiple faults.

This strategy focus identifying the set of $\mathcal{C}(T_{F_m})$, and as $T_{F_m} \cup T_{mask(F_m)} \supset T_{F_m}$, consequently, some schemas get through this strategy may be the parent-schema of some MFS. Moreover, some MFS may be irrelevant to the schemas get with this strategy, which means this strategy will ignore these MFS.

For the simple example in Table XII, when we using this strategy, we will get the minimal schemas (1, 1, -, 1) and (1, 1, 1, -), which are both the parent schemas of actual MFS (1,1,-,-), and we will observe that there is no schemas got by this strategy have any relationship with the actual MFS (-,1,1,1), which means it ignored the schema.

It is noted that, motivation example in section 2 actually adopted this strategy, which made the schemas identified for Ex 2: (-,2,4,5), (-,3,4,5) are the parent-schemas of the correct MFS(-,-,4,5).

4.2. masking effects for FCI approaches

With masking effects, the scenario of traditional FCI approaches is a bit complicated than previous two exhaustive testing scenario, which can be depicted in the Figure 4. In this figure, area A, C and D is the same as Figure 3, area B is divided into three sub-areas, in which B_1 is still the observed failing test cases for the current analysed fault, area B_2 is the test cases triggered other fault which masked the current fault and area B_3 is the test cases triggered other fault which did not mask the current fault.

With this model, if we have known which test cases mask the expect fault, i.e., figured out the B_2 and B_3 , then the schemas FCI approach will identify can be described as $\mathcal{C}(A \cup B_1 \cup B_2 \cup C)$. We next denote this result as *knowing masking effects*. However, as we discussed before, to get this result is not possible without human involvement. Correspondingly, when taking *regard as one fault* strategy, the MFS traditional FCI identify are $\mathcal{C}(A \cup B_1 \cup B_2 \cup B_3 \cup C)$. And for *distinguish faults* strategy, the MFS is $\mathcal{C}(A \cup B_1 \cup C)$. Next, we will respectively discuss the influence of masking effects on the two strategies.

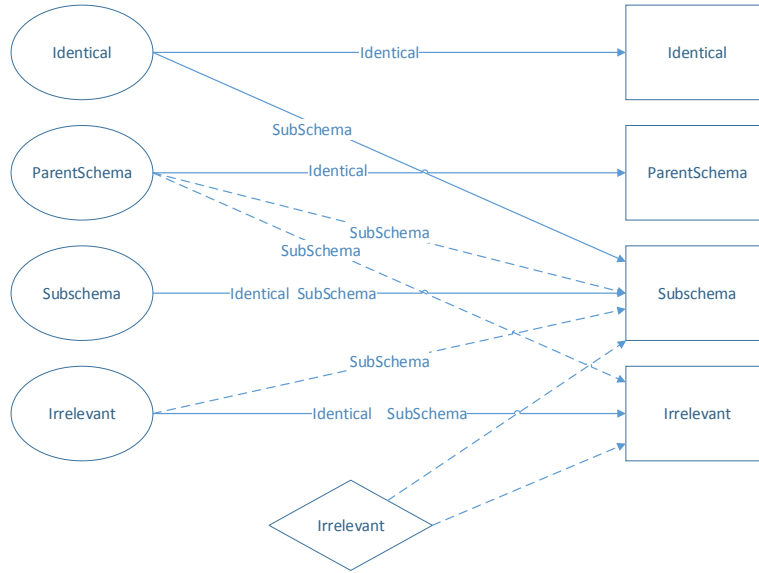


Fig. 5. masking effects influence on FCI with regard as one fault

4.2.1. Taking regard as one fault strategy. For the first strategy: *regard as one fault*, the impact of masking effects on FCI approaches can be described as Figure 5. To understand the content of this figure, let us get back to the relationship between the minimal schemas of two different set of test cases. For the *knowing masking effects* condition, the result identified by FCI approach, i.e., $\mathcal{C}(A \cup B_1 \cup B_2 \cup C)$, is one case of the *intersect* scenario. It means that the minimal schemas get in this condition can be identical, parent-schema, sub-schema, irrelevant to the actual MFS. And then apply the *regard as one fault*, the minimal schemas we got are $\mathcal{C}(A \cup B_1 \cup B_2 \cup B_3 \cup C)$. Obviously we have $A \cup B_1 \cup B_2 \cup B_3 \cup C \supset A \cup B_1 \cup B_2 \cup C$. So the minimal schema get by this strategy is either the sub-schema or identical to some schemas from the ones got by *known masking effects*, or exist some schemas irrelevant to all of them. Taking this two properties together, we will get Figure 5.

The ellipses in the left column of this figure illustrated the relation between the schemas identified under the *knowing masking effects* condition with the actual MFS, which includes the four possibilities: identical, sub-schema, parent-schema and irrelevant. For each relationship in this part, without of loss of generality, we consider c_{origin} and c_m , c_{origin} is the minima schema got by *knowing masking effects*, and c_m is the actual MFS. Then when we apply *regard as one fault* strategy, we may get some schemas which are identical or sub-schema of c_{origin} , say c_{new} s.t., $c_{new} = c_{origin}$ or $c_{new} \prec c_{origin}$. In this figure, we use the directed line to represent this two transformations with different labels in each line. As for the schemas c_{new} that is irrelevant to all the c_{origin} , we in the bottom of this figure use a rhombus labeled with "Irrelevant" to express them. Then we must have some relationship between the c_{new} and actual MFS, which also have four possibilities represented as rectangles in the right column.

Note that there exists two types of directed line, solid line and dashed line, in which the former indicates that this transformation is deterministic, e.g., if $c_{origin} \prec c_m$ and $c_{new} \prec c_{origin}$, then we must have $c_{new} \prec c_m$. The latter type means the transformation is just one of all the possibilities in such condition, e.g., if c_{origin} irrelevant c_m

and $c_{new} \prec c_{origin}$, then we can have either $\exists c_{m'}$ is one actual MFS, s.t. $c_{new} \prec c_m$ or c_{new} irrelevant to all actual MFS. (We will later give examples to illustrate them).

In this figure, only two transformation can make $\exists c_{m'}$ is one actual MFS, s.t. $c_{new} = c_{m'}$ or $c_{m'} \prec c_{new}$, which are when $c_{origin} = c_m$, $c_{new} = c_{origin}$ or $c_m \prec c_{origin}$, $c_{new} = c_{origin}$ respectively. This can be easily understood, as to make $c_{new} = c_{m'}$ or $c_{m'} \prec c_{new}$, according to the proposition 3.11 and 3.12, it must have $\mathcal{T}(c_{new}) \subset (B_1 \cup B_2)$. If after transformation, we got $c_{new} \prec c_{origin}$, then we must have $\exists t \in B_3$, s.t., $t \in \mathcal{T}(c_{new})$, otherwise, there should have no changing to the c_{origin} , and must have no newly schema c_{new} that is the sub-schema of c_{origin} . Consequently, $\mathcal{T}(c_{new}) \not\subset (B_1 \cup B_2)$ if we have $c_{new} \prec c_{origin}$. So in order to make $c_{new} = c_{m'}$ or $c_{m'} \prec c_{new}$, the transformation can only be identical, i.e., $c_{new} = c_{origin}$ and must have $c_{origin} = c_m$ or $c_m \prec c_{origin}$ correspondingly.

As the identical transformation is simple, so next we will ignore them and just take examples to illustrate the other transformations. Table XIII presents all these possibilities except these identical transformation. This table comprises of three parts, which the upper part give the test cases for each area in the abstract FCI model. It is noted that we merged the area B_1 , B_2 and C into one column as the way of computing the minimal schemas of three approaches – *actual MFS*, *knowing masking effects*, *regard as one fault* are all dependent on the merging test cases of this three area, not on the specific distribution of them. The middle part of this table shows the minimal schemas that using particular method. At last the bellow part in specific depicts the sample of each possible result of the transformation in the Figure 5. In this part, $c_m = c_{origin} \rightarrow c_{new} \prec c_{m'}$ indicates the scenario that the schema c_{origin} got by *knowing masking effects* is identical to one actual MFS c_m , then taking *regard as one fault*, we got $c_{new} \prec c_{origin}$, and such that we can have $\exists c_{m'}$ is one actual MFS, s.t., $c_{new} \prec c_m$. Other formula in this column can be explained in this way, in which c_{new} irrele and c_{origin} irrele means the schema c_{new} and c_{origin} is irrelevant to all the actual MFS respectively. The mark * in c_m and $c_{m'}$ respectively represent this two condition. The formula in the last two rows, $c_{new_{irrele}} \prec c_{m'}$ and $c_{new_{irrele}irrele}$ indicate the schemas $c_{new_{irrele}}$ got by *regard as one fault* is irrelevant to all the c_{origin} , in which the former is the subschema of some actual MFS $c_{m'}$ while the latter is irrelevant to all the actual MFS. The * in the c_{origin} column means that the c_{new} is irrelevant to all the c_{origin} .

4.2.2. using distinguish strategy. And for the second strategy: *distinguish faults*, the influence can be described as Figure 6.

This figure is organised the same way as figure 5. As with distinguish faults strategy, the minimal schemas identified are actually $\mathcal{C}(A \cup B_1 \cup C)$. Obviously $A \cup B_1 \cup C \subset A \cup B_1 \cup B_2 \cup C$. So under this transformation, the c_{new} should be either parent-schema or identical to the c_{origin} .

We take an example to illustrate this type of transformation, which is depicted in Table XIV. Similar with previous strategy, we omit the samples that is belong to the identical transformation.

It is noted that despite the transformations corresponding each directed line that is depicted in Figure 6, there is one more transformation can appear in this strategy, which can make some c_{origin} removed from the newly minimal schemas, i.e., $\neg c_{new}$, s.t., $c_{new} = c_{origin}$ or $c_{origin} \prec c_{new}$. For the example of the Table XIV, in the last row we used a formula c_{origin} ignored to represent this condition, the mark * in the c_{new} indicate that the c_{origin} is irrelevant to all the c_{new} . In this row, we can find for the schema $c_{origin} = (1, 1, 0, 0, 1, -)$, which is identical to the one in actual MFS, there exists no c_{new} which is identical or is the parent-schema of this schema. Consequently, in this condition, this strategy may ignore some actual MFS comparing with *knowing masking effects*.

Table XIII. Example the influence of regard as one fault for FCI approach

A	$B_1 \cup B_2 \cup C$	B_3	D	
(0,0,0,1,0,0)	(1,1,1,0,0,0)	(1,0,1,0,0,0)	(1,1,0,0,0,0)	
(0,0,0,1,1,0)	(1,1,1,0,1,0)	(1,0,1,0,1,0)	(1,1,0,0,1,0)	
(0,0,1,0,0,0)	(1,1,1,1,0,0)	(0,0,1,0,1,0)	(1,1,0,1,0,0)	
(0,0,1,1,0,0)	(1,1,1,1,1,0)	(0,0,1,1,1,0)	(1,1,0,1,1,0)	
	(1,0,1,1,0,0)	(0,1,0,0,0,0)	(0,0,1,1,0,1)	
	(1,0,1,1,1,0)	(0,1,0,1,0,0)	(0,1,1,1,0,1)	
	(0,0,0,0,1,0)	(0,1,1,0,0,0)		
	(0,0,0,0,0,0)	(0,1,1,1,0,0)		
	(0,0,1,1,1,1)	(1,0,0,0,0,0)		
	(0,1,1,1,1,1)	(1,0,0,0,1,0)		
		(1,1,1,1,1,1)		
		(1,0,1,1,1,1)		
<i>actual MFS</i> $\mathcal{C}(B_1 \cup B_2 \cup C \cup D)$	<i>knowing masking effects</i> $\mathcal{C}(A \cup B_1 \cup B_2 \cup C)$	<i>one fault</i> $\mathcal{C}(A \cup B_1 \cup B_2 \cup B_3 \cup C)$		
(1,1,-,-,0)	(1,1,1,-,-,0)	(1,-,1,-,-,0)		
(1,-,1,1,-,0)	(1,-,1,1,-,0)	(0,0,-,-,-,0)		
(0,0,0,0,-,0)	(0,0,0,-,-,0)	(0,-,-,-,0,0)		
(0,-,1,1,-,1)	(0,0,-,-,0,0)	(-,0,1,-,-,0)		
	(-,0,1,1,0,0)	(-,1,-,0,0,0)		
	(0,-,1,1,1,1)	(-,0,-,0,-,0)		
		(-,1,1,1,1,1)		
		(1,-,1,1,1,-)		
		(-,0,1,1,1,-)		
transformation	c_m	c_{origin}	c_{new}	$c_{m'}$
$c_m = c_{origin} \rightarrow c_{new} < c_{m'}$	(1,-,1,1,-,0)	(1,-,1,1,-,0)	(1,-,1,-,-,0)	(1,-,1,1,-,0)
$c_m < c_{origin} \rightarrow c_{new} < c_{m'}$	(1,1,-,-,-,0)	(1,1,1,-,-,0)	(1,-,1,-,-,0)	(1,-,1,1,-,0)
$c_m < c_{origin} \rightarrow c_{new} irrele$	(0,-,1,1,-,1)	(0,-,1,1,1,1)	(-,1,1,1,1,1)	*
$c_{origin} < c_m \rightarrow c_{new} < c_{m'}$	(0,0,0,0,-,0)	(0,0,0,-,-,0)	(0,0,-,-,-,0)	(0,0,0,0,-,0)
$c_{origin} irrele \rightarrow c_{new} < c_{m'}$	*	(0,0,-,-,0,0)	(0,0,-,-,-,0)	(0,0,0,0,-,0)
$c_{origin} irrele \rightarrow c_{new} irrele$	*	(-,0,1,1,0,0)	(-,0,1,-,-,0)	*
$c_{new} irrele < c_{m'}$	*	*	(-,0,-,0,-,0)	(0,0,0,0,-,0)
$c_{new} irrele$	*	*	(1,-,1,1,1,-)	*

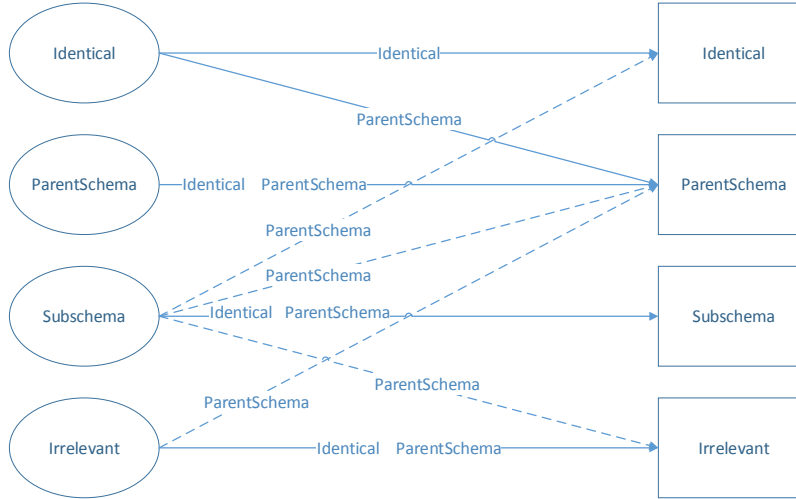


Fig. 6. masking effects influence on FCI with distinguish faults

Table XIV. Example the influence of distinguish faults for FCI approach

A	$B_1 \cup C$	B_2	D	
(0,0,0,1,1,0)	(0,0,0,0,0,0)	(0,0,1,1,1,0)	(0,1,1,0,0,0)	
(0,0,1,1,0,0)	(0,0,0,0,1,0)	(1,1,0,1,0,0)	(0,1,1,0,1,0)	
(0,0,1,0,1,0)	(0,0,0,1,0,0)	(1,0,1,1,0,1)	(0,1,1,1,0,0)	
(0,0,1,0,0,0)	(1,1,1,1,1,0)	(0,0,1,1,1,1)	(0,1,1,1,1,0)	
(1,1,0,0,0,0)	(1,1,0,0,1,0)	(1,1,0,0,1,1)	(1,1,1,1,1,1)	
(0,0,1,1,0,1)	(1,1,0,1,1,0)	(0,1,0,0,1,1)	(0,1,1,1,1,1)	
	(1,1,1,0,0,0)	(1,0,1,0,1,1)	(1,1,1,1,0,1)	
	(1,1,1,1,0,0)		(1,0,0,1,1,1)	
	(1,1,1,0,1,0)			
	(1,0,1,1,1,1)			
	(0,0,0,0,1,1)			
	(1,0,0,0,1,1)			
<i>actual MFS</i> $\mathcal{C}(B_1 \cup B_2 \cup C \cup D)$	<i>knowing masking effects</i> $\mathcal{C}(A \cup B_1 \cup B_2 \cup C)$	<i>distinguish faults</i> $\mathcal{C}(A \cup B_1 \cup C)$		
(0,-,1,1,1,-)	(1,1,-,-,0)	(0,0,0,-,-0)		
(1,1,-,1,-,0)	(0,0,-,-,-0)	(0,0,-,-,0,0)		
(-,0,0,1,1)	(-,0,1,1,-,1)	(0,0,-,0,-,0)		
(0,0,0,-,0,0)	(0,0,1,1,-,-)	(1,1,1,-,-,0)		
(0,0,0,0,-,0)	(0,0,0,0,1,-)	(1,1,-,-,1,0)		
(1,0,-,-,1,1)	(1,1,0,0,1,-)	(1,1,-,0,-,0)		
(1,1,0,0,1,-)	(1,0,1,-,1,1)	(0,0,1,1,0,-)		
(-,1,1,-,-0)	(1,0,-,0,1,1)	(1,0,1,1,1,1)		
(-,1,1,1,1)	(-,0,0,0,1,1)	(-,0,0,0,1,1)		
(1,1,-,-,1,0)		(0,0,0,0,1,-)		
(-,1,1,1,1,-)				
(1,1,1,1,-,-)				
(1,-,1,1,-,1)				
(0,0,0,0,1,-)				
transformation	c_m	c_{origin}	c_{new}	$c_{m'}$
$c_m = c_{origin} \rightarrow c_{m'} \prec c_{new}$	(-,0,0,1,1)	(-,0,0,1,1)	(-,0,0,0,1,1)	(-,0,0,0,1,1)
$c_m \prec c_{origin} \rightarrow c_{m'} \prec c_{new}$	(1,0,-,-,1,1)	(1,0,1,-,1,1)	((1,0,1,1,1,1))	((1,0,-,-,1,1))
$c_{origin} \prec c_m \rightarrow c_{new} \text{ irrele}$	(1,1,-,1,-,0)	(1,1,-,-,0)	(1,1,-,0,-,0)	*
$c_{origin} \prec c_m \rightarrow c_{new} \prec c_{m'}$	(0,0,0,0,-,0)	(0,0,-,-,-0)	(0,0,0,-,-0)	(0,0,0,0,-,0)
$c_{origin} \prec c_m \rightarrow c_{m'} \prec c_{new}$	(1,1,-,1,-,0)	(1,1,-,-,0)	(1,1,1,-,-0)	(-,1,1,-,-0)
$c_{origin} \prec c_m \rightarrow c_{m'} = c_{new}$	(1,1,-,-,1,0)	(1,1,-,-,-0)	(1,1,-,-,1,0)	(1,1,-,-,1,0)
$c_{origin} \text{ irrele} \rightarrow c_{m'} \prec c_{new}$	*	(-,0,1,1,-,1)	(1,0,1,1,1,1)	(1,-,1,1,-,1)
$c_{origin} \text{ irrele} \rightarrow c_{new} \text{ irrele}$	*	(0,0,1,1,-,-)	(0,0,1,1,0,-)	*
$c_{origin} \text{ ignored}$	(1,1,0,0,1,-)	(1,1,0,0,1,-)	*	*

In fact, besides this special case that may make the FCI approach ignore some actual MFS, there exists some other cases can also achieve the same effect, for example, when the c_{new} is only schema that is *related* to c_{origin} , (*related* means not irrelevant, in this case it is either identical or parent-schema). And the corresponding parent-schema c_{origin} is the only schema which is related to one actual MFS c_m . Then if the c_{new} is irrelevant to all the actual MFS, we will ignore the actual MFS c_m . This *ignored* event is caused of the c_{new} grow into the irrelevant schemas, which can also be appeared in the strategy *regard as one fault*. However, the beforehand cause of *ignored*—the c_{origin} is *removed* from the newly minimal schemas can only happens in the strategy *distinguish faults*.

4.3. Summary of the masking effects on FCI approach

From the analysis of formal model, we can learn that masking effects do influence the FCI approaches, worse more, both strategies *regard as one fault* and *distinguish faults* are harmful, which specifically the former comparing the knowing masking effects have a large possibility that get more sub-schemas of the actual MFS and get more schemas which are irrelevant to the MFS, while the latter may get more paren-

t schemas of the MFS and can also get more irrelevant MFS. Further, Both the two strategies can ignore the actual MFS and the *distinguish* is more likely ignore the MFS than the former strategy.

Note that our discussion is based on that SUT is a deterministic software, i.e., the random failing information of test case will be ignored. The non-deterministic problem will complex our test scenario, which will not be discussed in this paper.

5. TEST CASE REPLACING STRATEGY

The main reason why FCI approach fails to properly work is that we cannot determine the area B_2 and B_3 , i.e., if the test case under test trigger other faults which is different from the current one, we cannot figure out whether this test case will trigger the current expected fault as the masking effects may prevent that. So in order to limit the impact of this effect on FCI approach, we need to reduce the number of test cases that trigger other faults as much as possible.

In the exhaustive testing, as all the test cases will be used to identify the MFS, there is no room left to improve the performance without fixing the other faults and re-executing all the test cases. However, when just needing to select part of the whole test cases to identify MFS, which is as traditional FCI approach works, we can adjust the test cases we need to use by selecting proper ones so that we can limit the size of $\mathcal{T}(mask_{F_m})$ to be as small as possible.

5.1. Replace test case triggering unexpected fault

The basic idea is to pick the test cases that trigger other faults and generate newly test cases to replace them. These regenerated test cases should either pass in the execution or trigger F_m . The replacement must satisfy that the newly generated ones will not negatively influence the original identifying process.

Commonly, when we replace the test case that triggers unexpected fault with a new test case, we should keep some part in the original test case, we call this part as *fixed part*, and mutate other part with different values from the original one. For example, if a test case (1,1,1,1) triggered an unexpected fault, and the fixed part is (-,-,1,1), then we can replace it with a test case (0,0,1,1) which may either pass or trigger expected fault.

The *fixed part* can varies for different FCI approaches, e.g, for OFOT algorithm, the fixed part is the factors except for the one that need to be validated whether it is the component of the MFS, while for FIC_BS, we will fix the factors that should not be mutated of the test case in the next iteration of FIC_BS process.

It is noted that this replacement may need to execute multiple times for one fixed part as we could not always find a test case that by coincidence satisfy our requirement. Our previous work just randomly choose test cases until find one satisfied test case. That brute method may be simple and really works, but also may need to try too many times to get the satisfied one. So to handle this problem and reduce the cost, we augment our previous approach with computing the *strength* of the test case with the other faults, and then we will select one test case from a group of candidate test cases that has the least *strength* that is related to other faults.

To explain the *strength* notation, we need first introduce the *strength* that a factor is related to a particular fault. We use $all(o)$ to represent the number of executed test cases that contain this factor, and $m(o)$ to indicate the number of test cases that trigger the fault F_m and contain this factor. Then a the *strength* that a factor is related to a particular fault, i.e., $S(o, F_m)$, is $\frac{m(o)}{all(o)+1}$. This heuristic formula is based on the idea that if a factor frequently appears in the test cases that trigger the particular fault, then it is more likely to be the inducing factor that trigger this type of fault. We plus 1

in the denominator for two facts: 1. avoid the division by zero when the factor is never appears before, 2. reduce the bias when a factor rarely appears in the test set but by coincidence appears in a failing test case with a particular fault.

With this *strength* of the factor, we then define the *strength* a test case f is related to a particular fault F_m as:

$$S(f, F_m) = \frac{1}{k} \sum_{o \in f} S(o, F_m)$$

In this formula, k is the number of factors in the test case f , o is the specific factor in f . This formula computes the average *strength* of all the factors in the test case as the *strength* for this test case that is related to a particular fault. For a test case that is selected to be tested, we need to make it as small possibility to trigger other faults. In practice, one test case can have different related *strength* to different faults, so we can not always find a test case that have the least relating *strength* to all the faults when comparing other test cases. With this fact in mind, our target changes to find a test case that the maximal *strength* it is related fault among other faults should be the minimal comparing other test cases. Formally, we should choose a test case f , s.t.,

$$\min_{f \in R} \max_{m \leq L \& m \neq n} S(f, F_m) \quad (\text{EQ1})$$

In this formula, L is the number of all the faults, n is the current analysed fault. R is the set of all the possible test cases that contain the *fixed* part except those ones that have been tested. Obviously $|R| = \prod_{i \notin \text{fixed}} (v_i) - |t \mid t \text{ contains fixed part \& } t \text{ is tested}|$.

We can further resolve this problem. Consider the test case we get – f satisfied the EQ1. Without loss of generality, we assume that the fault $F_k, k \neq n$ is the fault that the test case f has the maximal related *strength* than other faults. Then a natural property for f is that any other test case f' which satisfies that fault F_k is the maximal related fault for this test case must have $S(f, F_k) \leq S(f', F_k)$. Formally, to get such a test case is to solve the following formula :

$$\begin{aligned} \min \quad & S(f, F_k) \\ \text{s.t.} \quad & f \in R \\ & S(f, F_k) > S(f, F_i), \quad 1 \leq i \leq L \& i \neq k, n \end{aligned} \quad (\text{EQ2})$$

With this formula, to solve the EQ1, we just need to find the particular fault F_k , such that the related *strength* between the test case f satisfies the EQ2 and this fault is the smallest than other faults. Formally, we need to find :

$$\begin{aligned} \min \quad & S(f, F_k) \\ \text{s.t.} \quad & 1 \leq k \leq L \& k \neq n \\ & f, F_k \text{ satisfies EQ2} \end{aligned} \quad (\text{EQ3})$$

According to the EQ3, the problem that to get such a test case lies on to solve the EQ2 because if that is solved we just need to rank the one has the minimal value from the solutions to EQ2. As to EQ2, it can be formulated as an 0-1 integer linear programming (ILP) problem. Assume the SUT we test have K factors, in which the i th factor has V_i values can take from. And the SUT has L faults. We then define the variable x_{ij} as:

$$x_{ij} = \begin{cases} 1 & \text{the } i\text{th factor of the test case take the } j\text{th value for that factor} \\ 0 & \text{otherwise} \end{cases}$$

We then take $o_{m_{ij}}$ to be the related *strength* between the j th value of the i th factor of the SUT and the fault F_m . And we use a set R of factors with its values to define the fixed part in the test case we should not change, i.e., $R = \{(i, j) | i \text{ is the fixed factor in the test case, } j \text{ is the corresponding value}\}$. As we can generate redundant test cases, so we keep a set of test cases $T_{executed}$ to guide to generate different test cases. Then the EQ2 can be detailed as the following ILP:

$$\min \quad \frac{1}{|K|} \sum_{i=0}^K \sum_{j=0}^{V_i} o_{m_{ij}} \times x_{ij} \quad (\text{EQ4})$$

$$\text{s.t.} \quad 0 \leq x_{ij} \leq 1 \quad i = 0..K-1, j = 0, ..V_i-1 \quad (1)$$

$$x_{ij} \in \mathbb{Z} \quad i = 0..K-1, j = 0, ..V_i-1 \quad (2)$$

$$\sum_{j=0}^{V_j} x_{ij} = 1 \quad i = 0..K-1 \quad (3)$$

$$x_{ij} = 1 \quad (i, j) \in R \quad (4)$$

$$\sum_{i=0}^K \sum_{j=0}^{V_i} (o_{m_{ij}} - o_{m'_{ij}}) \times x_{ij} \geq 0 \quad 1 \leq m' \neq m \leq L \quad (5)$$

$$\sum_{(i,j) \in t} x_{ij} < K \quad t \in T_{existed} \quad (6)$$

In this formula, the constraint (1) and (2) indicate that the variable x_{ij} is a 0-1 integer. Constraint (3) indicates that a factor in one test case can only take one value. Constraint (4) indicates the test case should not change values to the fixed part. Constraint (5) indicates that the related strength between Fault F_m and the test case is maximal than other. Constraint (6) indicates the test cases generated should not be the same as the test cases in $T_{existed}$.

As we have formulated the problem into a 0-1 integer programming problem, we just need to utilize a ILP solver to solve this formula. In this paper, we use a the solver introduced in [Berkelaar et al. 2004], which is a mixed Integer Linear Programming (MILP) solver can handle satisfaction and optimization problems.

The complete process of replacing a test case with a new one keeping some fixed part is depicted in Algorithm 1:

The inputs for this algorithm consists of the fault type we currently focus on – F_m , the fixed part which we want to keep from the original test case – s_{fixed} , the values sets that each factor can take from respectively – $Param$ and the set of matrix o_1, \dots, o_L , for any element in which, say o_m , record the related strength between each specific factor with each value and the fault F_m , i.e., $o_m = \{o_{m_{ij}} | 0 \leq i \leq K-1, 0 \leq j \leq V_i\}$. The output of this algorithm is a test case t_{new} which either trigger the expected F_m or passes.

This algorithm is a outer loop(line 1 - 19) containing two parts:

The first part(line 2 - 9) generates a new test case which is supposed to be least likely to trigger faults different from F_m . The basic idea for this part is to search each fault different from F_m (line 3) and find the best test case that has the least related strength with other faults. In detail, for each fault we setup an ILP solver (line 4) and use it to get an optimal test case for that fault according to EQ4 (line 5). We compare the optimal value for each fault, and choose the one has an less strength related to other faults (line 6 - 9).

ALGORITHM 1: replace test cases triggering unexpected faults

Input: fault type F_m , fixed part s_{fixed} , values set that each option can take $Param$, the related strength matrix $o_1 \dots o_L$

Output: t_{new} the regenerate test case, The frequency number

```

1 while not MeetEndCriteria() do
2    $optimal \leftarrow MAX$ ;  $t_{new} \leftarrow null$ ;
3   forall the  $F_k \in F_1, \dots, F_m, F_{m+1} \dots F_L$  do
4      $solver \leftarrow setup(s_{fixed}, Param, F_m, o_1 \dots o_L)$ ;
5      $(optimal', t'_{new}) \leftarrow solver.getOptimalTest()$ ;
6     if  $optimal' < optimal$  then
7        $t_{new} \leftarrow t'_{new}$ ;
8     end
9   end
10   $result \leftarrow execute(t_{new})$ ;
11   $updateRelatedStrengthMatrix(t_{new})$ ;
12  if  $result == PASS$  or  $result == F_m$  then
13    return  $t_{new}$ ;
14  else
15    continue;
16  end
17 end
18 return  $null$ 

```

Second part is to check whether the newly generated test case is as expected (line 10 - 16). We first execute the SUT under the newly generated test case (line 10) and update the related strength matrix ($o_1 \dots o_L$) for each factor that is involved in this newly generated test case (line 11). We then check the executed result: if either the test case passes or triggers the same fault – F_m , we will get an satisfied test case (line 12), and we will directly return this test case (line 13). Otherwise, we will repeat the process, i.e., generate newly test case and check again (line 14 - 15).

It is noted that this algorithm has another exit besides we find an expected test case (line 12), which is when the function *MeetEndCriteria()* returns *true* (line 1). We didn't explicitly show what the function *MeetEndCriteria()* is like, because this is depending on the computing resource and how accurate you want to the identifying result to be. In detail, if you want to get a high quality result and you have enough computing resource, you can try much times to get the expected test case, otherwise, a relatively small number of attempts is recommended.

In this paper, we just set 3 as the biggest repeated times for this function. When it ended with *MeetEndCriteria()* is true, we will return null (line 18), which means we cannot find an expected test case.

5.2. A case study with the replacing strategy

Suppose we have to test a system with eight parameters, each has three options. And when we execute the test case $T_0 = (0\ 0\ 0\ 0\ 0\ 0\ 0\ 0)$, a failure – $e1$ is triggered. Next we will use the FCI approach – FIC_BS [Zhang and Zhang 2011] with replacing strategy to identify the MFS for the $e1$. Furthermore, there are two more potential faults $e2$ and $e3$ may be triggered during the testing, which will mask the desired fault $e1$. The process is listed in Figure 7. In this figure, there is two main columns in this figure, the left main column indicates the executed test cases during testing as well as their executed result, and each executed test case corresponds to a specific label $T_1 - T_8$ in the left. The right main column lists the related strength matrix when we comes with a test case triggered $e2$ or $e3$. In detail, the matrix records the related strength between

Table XV. The number of test cases each FCI approach needed to identify MFS

Method	number of test cases to identify MFS
Charles ELA	depends on the covering array
Martinez with safe value [Martínez et al. 2008; 2009]	$O(d \log k + d^2)$
Martinez without safe values [Martínez et al. 2008; 2009]	$O(d^2 + d \log k + \log^c k)$
Martinez' ELA [Martínez et al. 2008; 2009]	$ds^v \log k$
Shi SOFOT [Shi et al. 2005]	k
Nie OFOT [Nie and Leung 2011a]	kd
Yilmaz classification tree	depends on the covering array
FIC [Zhang and Zhang 2011]	k
FIC_BS [Zhang and Zhang 2011]	$t(\log k + 1) + 1$
Ghandehari's suspicious based [Ghandehari et al. 2012]	depends on the number and size of MFS
TRT [Niu et al. 2013]	$d \times t \times \log k + t^d$

each factor (column $O1 - O8$) for each value it can take (column v) with the unexpected fault (column F). The executed test case which is in bold indicates the one that trigger other faults and should be replaced in the next iteration.

From Figure 7, for the test case that triggered $e2 - (2\ 1\ 1\ 1\ 0\ 0\ 0\ 0)$ (in this case, the fixed part of the test case is $(- - - 0\ 0\ 0\ 0)$, which the last four factors is the same as the original test case T_0), we generates the related matrix in the left. Each element in this matrix is computed as the $\frac{m(o)}{all(o)+1}$, for example, for the $O7$ factor with value 0, we can find two test cases contain this element, i.e., T_0 and T_1 , so the $all(o)$ is 2. And only one test case trigger the fault $e2$, which means $m(o) = 1$. So the final related strength between this factor with $e2$ is $\frac{1}{2+1} = 0.33$. All the related strength with $e3$ is labeled with a short slash as there is no test case triggering this fault in this iteration. After this matrix is determined, we can get a optimal test case with ILP solver, which is $T'_1 - (1\ 2\ 2\ 2\ 0\ 0\ 0\ 0)$, with its related strength 0.167, which is smaller than others.

This replace process triggered each time the newly test case trigger other fault, and until we finally get the MFS. Some times we could not find a satisfied replacing test case in just one trial like T_1 to T'_1 . When it happened, we will need repeat searching the proper test case we desired, e.g., For example, for T_4 which triggered $e3$, we tried three times— T'_4, T''_4, T'''_4 to finally get a satisfied one T''''_4 which passes the testing. It is noted that the matrix continues to change with the test case generated and executed so that we can adaptively find a optimal one in the current process.

5.3. Complexity analysis

This complexity lay on two facts: the number of test cases triggered other faults which need to be replaced, and the number of test cases that need to be tried to generate a non-masking-effects test case. The complexity is the product of this two facts.

The first fact is comparable to the extra test cases that needed to identify the MFS, and this number varies in different FCI approaches. Table XV list the number of test cases that each algorithms that needed to get the MFS. In this table, d indicates the number of MFS in the SUT. k means the number of the parameters of the SUT. t is the number of factors of the MFS in the SUT. c is an upper bond, satisfies, $d \leq \frac{c}{2} \log \log k$. v is the number of values one parameter can take.

It must be noted that each algorithm may be limited to some restrictions to identify the result, details can be saw in [Zhang and Zhang 2011].

To get the magnitude of the second fact, we need to figure out the possibility that a test case that could trigger other fault. The first thing we need to care is the *fixed* part, as the additional generated test case should somehow contain this part. As we have mentioned before we can in fact generate $(v - 1)^{k-p}$ (p is the number of factors in the *fixed* part) possible test cases that contain the *fixed* part. Apart from the one that need to be replaced, there remain $(v - 1)^{k-p} - 1$, which indicates the complexity is

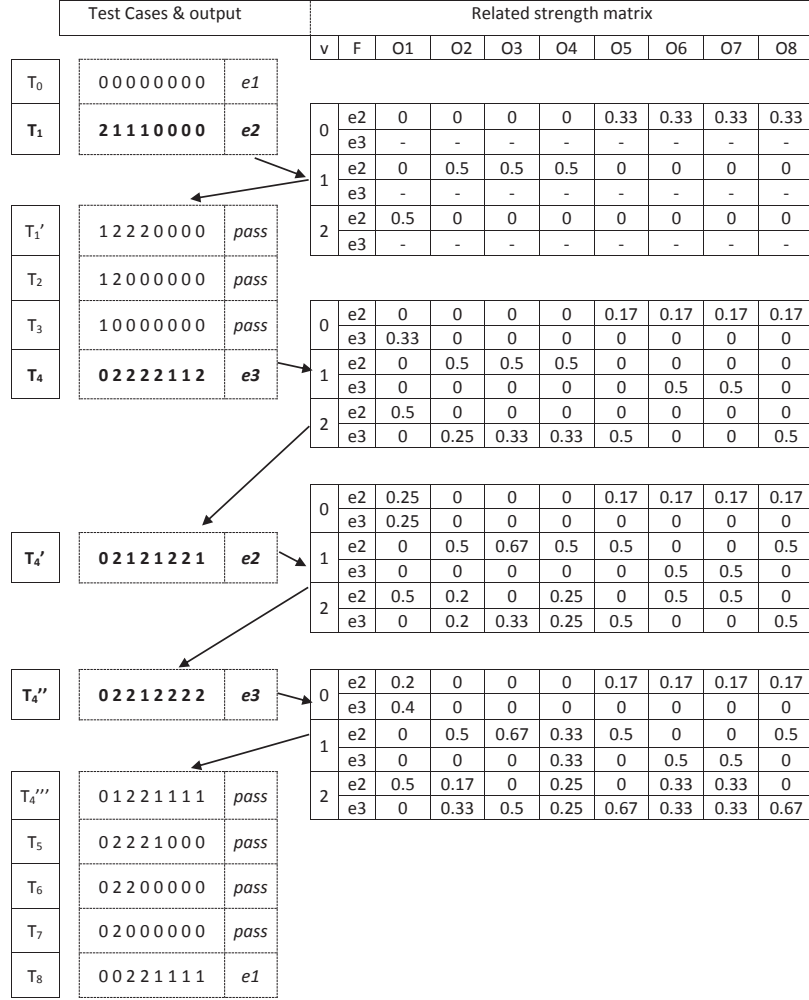


Fig. 7. A case study of our approach

$O((v-1)^{k-p} - 1)$. However, to avoid the exponential computational complexity, we in this algorithm use a method *MeetEndCriteria()* (line 1) function to end the algorithm when the trying times is over a prior given constant, say N , so the final complexity for the second part is $O(\min(N, (v-1)^{k-p} - 1))$.

It is noted that exponential factor $k-p$ directly affects the complexity of the second factor, the more p is, the less test cases could be generated. For different approach, p is different. As for OFOT, p is a fixed number, which is $k-1$, it is the minimal number of the test cases we need to give. While for FIC_BS, the p varies in the test cases it generates, ranges from $k-1$ to 1. While for non-adaptive, as the *fixed* part is the coverage, we list them as t . We list them all in Table XVI. It is noted that the *Martinezwithoutsafevalues* have no such complexity, this is because this approach works when $v=2$, and this will result in that we will not have other test cases to be replaced if we test a fixed part when trigger other faults.

Table XVI. The complexity of second part

Method	fixed part
Charles ELA	$O(\min(N, (v-t)^{k-p} - 1))$
Martinez with safe value [Martínez et al. 2008; 2009]	$O(\min(N, (v-1) - 1)) O(\min(N, (v-1)^{k-1} - 1))$
Martinez without safe values [Martínez et al. 2008; 2009]	—
Martinez' ELA [Martínez et al. 2008; 2009]	$O(\min(N, (v-t)^{k-p} - 1))$
Shi SOFOT [Shi et al. 2005]	$O(\min(N, (v-1) - 1))$
Nie OFOT [Nie and Leung 2011a]	$O(\min(N, (v-1) - 1))$
Yilmaz classification tree	$O(\min(N, (v-t)^{k-p} - 1))$
FIC [Zhang and Zhang 2011]	$O(\min(N, (v-1) - 1)) O(\min(N, (v-1)^{k-1} - 1))$
FIC.BS [Zhang and Zhang 2011]	$O(\min(N, (v-1) - 1)) O(\min(N, (v-1)^{k-1} - 1))$
Ghandehari's suspicious based [Ghandehari et al. 2012]	$O(\min(N, (v-t)^{k-p} - 1))$
TRT [Niu et al. 2013]	$O(\min(N, (v-1) - 1)) O(\min(N, (v-1)^{k-1} - 1))$

Table XVII. Software under survey

software	versions	LOC	classes	bug pairs
HSQLDB	2.0rc8	139425	495	#981 & #1005
	2.2.5	156066	508	#1173 & #1179
	2.2.9	162784	525	#1286 & #1280
JFlex	1.4.1	10040	58	#87 & #80
	1.4.2	10745	61	#98 & #93

6. EMPIRICAL STUDIES

To investigate the impact of masking effects for FCI approaches in real software testing scenario and to evaluate the performance that how well our approach handle this effect, we conduct several empirical studies in this section. Each one of the studies focus on addressing one particular problem, which are listed as following:

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

Q2: How much do our approach performs compared with traditional approaches?

Q3: Is the ILP based test case searching technique efficient compared with randomly selecting?

Q4: Comparing with existed masking effects handling work – FDA-CIT [Yilmaz et al. 2013], do our new approach have any advantages ?

6.1. The existence and characteristics of masking effects

In the first study, we surveyed two open-source software to gain an insight on the existence of multiple faults and their effects. The software under study are: HSQLDB and JFlex, the first is a database management software written in pure java and the second is a lexical analyser generator. Each of them contain different versions. All the two 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 XVII lists the program, the number of versions we surveyed, number of lines of uncommented code, number of classes in the project and the bug's id ³ of each software we studied.

6.1.1. Study setup. We first looked through the bug tracker forum of each software and focus on the bugs which are caused by the options combination. For each such bug, we will derive its MFS 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#981 for HSQLDB, we found the failure-inducing combinations for

³<http://sourceforge.net/p/hsqldb/bugs>
<http://sourceforge.net/p/jflex/bugs>

this bug is: (*preparestatement*, *placeholder*, *Long string*), this three factors together form the condition on which the bug will be triggered. These analysed results will be regarded as the “prior MFS” later.

We further built the testing scenario for each version of the software listed in Table XVII. The testing scenario is properly constructed so that we can reproduce different faults through controlling the inputs to that test file. For each particular version of the software, the source code of the testing file as well as other detailed experiment information is available at— <https://code.google.com/p/merging-bug-file>.

Next we built the input model which consist of the options related to the failure-inducing combinations and additional noise options. The detailed model information is in Table XVIII and XIX for HSQLDB and JFlex respectively. Each table is organised into four groups: 1) “common options”, which lists the options as well as their values under which every version of this software can be tested. 2) “common boolean options”, which lists additional common options whose data type is boolean. 3) “specific options”, under which only the specific version of that software can be tested. 4) “configure space”, which depicts the input model for each version of the software, the input model is presented in the abbreviated form $\#values^{\#number\ of\ parameters} \times \dots$, e.g., $2^9 \times 3^2 \times 4^1$ indicates the software have 9 parameters that can take 2 value, 2 parameters can take 3 values and only one factor can take 4 values.

Table XVIII. Input model of HSQLDB

common options		values
Server Type	existed form	server, webserver, inprocess mem, file
resultSetTypes		forwad, insensitive, sensitive
resultSetConcurrencys		read_only, updatable
resultSetHoldabilitys		hold, close
StatementType		statement, prepared
common boolean options		
sql.enforce_strict_size, sql.enforce_names, sql.enforce_refs		
versions	specific options	values
2.0rc8	more	true, false
	placeholder	true, false
	cursorAction	next,previous,first,last
2.2.5	multiple	one, multi, default
	placeholder	true, false
2.2.9	duplicate	dup, single, default
	default.commit	true, false
versions	Config space	
2.0rc8	$2^9 \times 3^2 \times 4^1$	
2.2.5	$2^8 \times 3^3$	
2.2.9	$2^8 \times 3^3$	

We then generated the exhaustive test suite consist of all the possible combinations of these options and under each of them we executed the prepared testing file. We recorded the output of each test case to observe whether there are test cases contain prior MFS but do not produce the corresponding bug.

6.1.2. Result and discussion. Table XX lists the results of our survey. Column “all tests” give the total number of test cases we executed, Column “failure” indicate the number of test cases that failed during testing and Column “masking” indicate the number of test cases which trigger the masking effect.

We observed that for each version of the software under analysis we listed in the Table XX, the test cases with masking effects do exist, i.e., test cases containing MFS did not trigger the corresponding bug. In effect, there is about 768 out of 4608 test

Table XIX. Input model of JFlex

common options		values
generation		switch, table, pack
charset		default, 7bit, 8bit, 16bit
common boolean options		
public, apiprivate, cup, caseless, char, line, column, notunix, yyeof		
versions	specific options	values
1.4.1	hasReturn	has, non, default
	normal	true, false
1.4.2	lookAhead	one, multi, default
	type	true, false
	standalone	true, false
versions	Config space	
1.4.1	$2^{10} \times 3^2 \times 4^1$	
1.4.2	$2^{11} \times 3^2 \times 4^1$	

Table XX. Number of faults and their masking effects

software	versions	all tests	failure	masking
HSQldb	2cr8	18432	4608	768
-	2.2.5	6912	3456	576
-	2.2.9	6912	3456	1728
JFlex	1.4.1	36864	24576	6144
-	1.4.2	73728	36864	6144

cases (16.7%) in hsqldb with 2rc8 version. This rate is about 16.7%, 50%, 25%, 16.7% respectively for the remaining software versions, which is not trivial.

So the answer to **Q1** is that in practice, when SUT have multiple faults, the masking effects do exist widely in the test cases.

6.2. Compare our approach with traditional algorithms

In the second study, our aim is to compare the performance of identifying the MFS under the impact of masking effects between our approach with traditional ones. To conduct this study, we need to respectively apply the our approach and traditional algorithms on identifying MFS in a group of software and evaluate their identifying results. The five prepared versions of software in Table XVII as testing subjects is far from a general evaluation subjects, however, to construct such real testing scenarios is time-consuming as we need to carefully study the tutorial of that software as well as the bug tracker report. So in order to give a desirable result based on more testing subjects, we then synthesize a bunch of such testing scenarios of which the characterization such as the number of factors, the number of faults and the possible masking effects are similar to the real software. In detail, we set the number of parameters k of the SUT ranged from 8–30. We limit the scale of the SUT to a relatively small size because we need to exhaustive execute each possible test case of the SUT to select the failing test cases and then fed them to the FCI approach. We then randomly choose 10 such SUTs and for each SUT we inject 2–5 different MFS into it, which can mask each other. The degree of the MFS we injected are ranged from 1 to 6.

Above all, Table XXI lists the testing model of both the real and synthesizing testing scenario. In this table, the column ‘software’ indicate the SUT under test, for the real SUT, we label it with the form ‘*name + version*’, while for the synthesizing ones, we label them as ‘*synthes+ id*’. The column ‘Model’ presents the model of the input space for that software. Last column shows the MFS as well as their masking sequence for each testing subject. The MFS is presented in an abbreviated form $\{\#index\#value\}$, e.g., $(5_1, 6_0, 7_0)$ actually means $(- - - - 1, 0, 0, -, -, -, -)$ for HSQldb of version ‘2cr8’. It

Table XXI. The testing models of the case study

software	Model	MFS& masking sequence
HSQLDB 2cr8	$2^9 \times 3^2 \times 4^1$	$(5_1, 6_0, 7_0) \rightarrow (5_1, 8_2, 9_2) = (5_1, 8_2, 9_1) \rightarrow (5_1, 8_3, 9_2) = (5_1, 8_3, 9_1)$
HSQLDB 2.2.5	$2^8 \times 3^3$	$(6_1, 7_0) \rightarrow (5_2)$
HSQLDB 2.2.9	$2^8 \times 3^3$	$(6_0) \rightarrow (0_1, 5_1, 7_0) = (0_0, 5_1, 7_0) \rightarrow (5_1, 7_0)$
JFlex 1.4.1	$2^{10} \times 3^2 \times 4^1$	$(0_0) \rightarrow (1_0)$
JFlex 1.4.2	$2^{11} \times 3^2 \times 4^1$	$(1_0, 2_1) \rightarrow (0_1)$
synthez 1	$2^5 \times 3^3 \times 4^1$	$(2_1, 3_0) \rightarrow (1_1, 2_1) = (1_0, 3_0)$
synthez 2	$2^6 \times 3^2 \times 4^1$	$(4_1, 6_0, 7_1, 8_0) \rightarrow (1_1, 3_1, 5_1) \rightarrow (2_0, 3_1, 6_0)$
synthez 3	$2^5 \times 3^3$	$(2_1, 3_0) \rightarrow (1_0) = (4_1) \rightarrow (6_0, 7_0)$
synthez 4	$2^7 \times 3^2 \times 4^1$	$(0_1, 2_1, 5_0, 6_1) \rightarrow (2_1, 4_0) = (6_1, 7_0) \rightarrow (3_0, 4_0, 5_0)$
synthez 5	$2^4 \times 3^3 \times 4^2$	$(0_0, 1_1, 3_0, 6_1, 8_0) \rightarrow (2_0, 3_0, 4_1)$
synthez 6	$2^9 \times 3^2$	$(2_0, 7_1, 8_1) \rightarrow (3_1, 5_1) = (4_0) \rightarrow (3_1, 6_0, 7_1) \rightarrow (3_1, 7_1, 8_0)$
synthez 7	$2^{10} \times 3^1 \times 4^1$	$(3_1, 4_0, 5_0) \rightarrow (2_0, 4_0, 7_1, 9_0) \rightarrow (6_1, 10_0, 11_1)$
synthez 8	$2^{11} \times 3^1 \times 4^1$	$(1_0, 3_1, 4_0, 7_1, 9_0, 12_1) \rightarrow (0_0, 2_1, 3_1, 7_1, 10_0, 11_1)$
synthez 9	$2^4 \times 4^3$	$(3_1, 5_0) \rightarrow (5_0, 6_1)$
synthez 10	$2^7 \times 3^3 \times 4^1$	$(0_1, 3_0, 4_1, 7_0) \rightarrow (2_0, 3_0, 5_1) = (2_0, 3_0, 5_0)$

is noted that we using ' \rightarrow ' and ' $=$ ' to describe the masking sequence of each MFS, in which ' \rightarrow ' means the left MFS in this operator can mask the right MFS of this operator, e.g., $(5_1, 6_0, 7_0) \rightarrow (5_1, 8_2, 9_2)$ means if $(5_1, 6_0, 7_0)$ appears in the test case, then $(5_1, 8_2, 9_2)$ will not be triggered. Operator ' $=$ ' means that this two MFS will not mask each other.

6.2.1. Study setup. After preparing the subjects under testing, we then apply our approach (augment the FIC_BS with replacing strategy) on identifying the MFS of each SUT listed in Table XXI. In specific, for each SUT we select each test case that failing during testing and feed them to our FCI approach as the input, then after the identifying process is over, we record the MFS it got (refer as *identified MFS* for convenience) and the extra test cases it needed. For traditional FIC_BS approach we design the same experiment as our approach, but as the subject under test has multiple faults for which the traditional FIC_BS can not directly applied on, we adopted two traditional strategies on the FIC_BS algorithm, i.e., *regard as one fault* and *distinguish faults* described in Section 3.2. The purpose of recording the generated additional test cases is to later quantify the additive cost of our approach.

We next compared the identified MFS of each approach with the prior MFS to quantify the degree that each suffers from masking effect, such that we can figure out how much our approaches performs better than traditional ones when the SUT contain potential masking effects. There are five metrics we need to care in this study, which are listed as follows:

- (1) The number of the common combinations appeared in both identified MFS and prior MFS. We denote it as *accurate number* later.
- (2) The number of the identified combinations which is the parent combinations of some prior failure-inducing combinations. We refer it to *parent number*.
- (3) The number of the identified combinations that is the sub combinations of some prior failure-inducing combinations, which is referred to *sub number*.
- (4) The number of ignored failure-inducing combinations. This metric counts these combinations in prior failure-inducing combinations, which are irrelevant to the identified combinations. We label it as *ignored number*.

- (5) The number of irrelevant combinations. This metric counts these combinations in these identified combinations, which are irrelevant to the prior failure-inducing combinations. It is referred to the *irrelevant number*.

Among these five metrics, high *accurate number* value indicates FCI approaches performs effectively, while *ignored number* and *irrelevant number* indicate the degree of deviation for the FCI approaches. For *parent number* and *sub number*, they indicate FCI approaches that, although with additional noisy information, can determine part parameter values about the failure-inducing combinations.

Besides these specific metric, we in addition give an composite criteria to measure the overall performance of each approach. The computing formula for this composite criteria is as follows:

$$\frac{accurate + related(parent) + related(sub)}{accurate + parent + sub + irrelevant + ignored}$$

In this formula, *accurate*, *parent*, *sub*, *irrelevant*, *ignored* respectively represents the value of each specific metric. *related* function gives the similarity between the schemas (either parent or sub) and the real MFS. The similarity between two schemas S_A and S_B is computed as:

$$Similarity(S_A, S_B) = \frac{\text{the same elements in } S_A \text{ and } S_B}{\max(Degree(S_A), Degree(S_B))}$$

For example, the similarity of (- 1 2 - 3) and (- 2 2 - 3) is $\frac{2}{3}$, as the same elements of this two schemas are the third and last elements, and both of this two schemas is 3-degree.

So the *related* function is the summation of similarity of all the parent or sub schemas with their corresponding MFS.

6.2.2. Result and discussion. TableXXII depicts the result of the second case study. There are eight main columns in this table, (too wide so we break it into two parts), which respectively indicates: the subject we apply FCI approach, the number of accurate MFS each approach identified, the number of identified schemas which is the sub-schema parent-schema of some prior MFS, The number ignored prior MFS, the number of identified schemas which is irrelevant to all the prior MFS, the metric which gives the overall evaluation of each approach and the extra test cases each algorithm needed. For each main colum, there are four sub-columns which respectively depicts the result for the FCI with *regard as one fault* strategy, FCI with *distinguish faults* strategy, FCI with replacing strategy based on ILP test case searching and FCI with replacing strategy based on randomly searching. The last one will be discussed in the next case study.

We first observed that, the results of two traditional strategies *regard as one fault* and *distinguish faults* are coincide with the formal analysis in section 4. In specific, the former has more sub-schemas of MFS than the latter for all the 15 subjects, and the latter has more parent-schemas of MFS that the former. For the ‘ignored MFS’ metric, as we have executed all the failing test cases, we get 0 ignored MFS for all the approaches. So in order to evaluate this metric of the approaches, we record this metric for each FCI one test case one time and take the average value as the result, which is listed in the parentheses. We also found in most cases (except *synthex 5* and *synthex 6*) the approach with *distinguish* strategy get more ignored MFS than another one, which is also coincide with the formal analysis. And for the ‘irrelevant MFS’, we found *regard as one fault* strategy in most cases get more ‘irrelevant MFS’ than another, which is also as expected.

Table XXII. Result of the evaluation of each approach

Subject	accurate				sub				parent				ignore			
	One	Distin	ILP	Rand	One	Distin	ILP	Rand	One	Distin	ILP	Rand	One	Distin	ILP	Rand
HSQL2cr8	2	3	5	5	3	2	0	2	0	2	4	5	0(2.04)	0(3.91)	0(4.0)	0(4.1)
HSQL2.2.5	2	2	2	2	1	0	0	0	0	1	1	1	0(0.83)	0(1.0)	0(1.0)	0(1.0)
HSQL2.2.9	2	3	2	2	3	1	1	1	0	4	1	1	0(1.33)	0(2.83)	0(2.56)	0(2.56)
JFlex1.4.1	2	2	2	2	0	0	0	0	0	1	0	0	0(0.75)	0(1.0)	0(1.0)	0(1.0)
JFlex 1.4.2	2	2	2	2	1	0	0	0	0	1	1	1	0(0.67)	0(1.0)	0(1.0)	0(1.0)
synthes 1	2	1	1	1	2	0	0	0	0	2	2	2	0(1.0)	0(2.0)	0(2.0)	0(2.0)
synthes 2	3	3	3	3	10	0	0	0	0	10	6	6	0(1.96)	0(2.0)	0(2.0)	0(2.0)
synthes 3	4	3	3	3	4	2	2	2	0	5	5	5	0(2.08)	0(2.84)	0(2.84)	0(2.84)
synthes 4	3	3	3	3	10	3	2	3	0	6	5	5	0(2.6)	0(2.8)	0(2.9)	0(2.85)
synthes 5	2	2	2	2	4	0	0	0	0	2	1	1	0(1.02)	0(1.0)	0(1.0)	0(1.0)
synthes 6	2	3	3	3	15	4	4	4	0	8	8	8	0(1.99)	0(3.72)	0(3.72)	0(3.72)
synthes 7	3	3	3	3	10	0	0	0	0	6	6	6	0(2.04)	0(2.0)	0(2.0)	0(2.0)
synthes 8	2	2	2	2	4	0	0	9	0	4	3	3	0(1.05)	0(1.0)	0(1.0)	0(1.0)
synthes 9	2	1	1	1	1	0	0	0	0	1	1	1	0(0.8)	0(1.0)	0(1.0)	0(1.0)
synthes 10	0	1	1	1	3	1	1	1	0	0	0	0	0(1.0)	0(1.46)	0(1.31)	0(1.31)

irrelevant				overall				test cases			
One	Distin	ILP	Rando	One	Distin	ILP	Rand	One	Distin	ILP	Rand
0	2	0	34	0.57	0.65	0.88	0.23	8.125	11.92	17	17.72
7	0	0	0	0.25	0.83	0.83	0.83	8.67	7.67	10.17	11.3
4	0	0	0	0.4	0.74	0.8	0.8	9.167	8.61	11.72	13.14
25	0	0	0	0.07	0.83	1	1	23.5	6.5	8	9.68
25	0	0	0	0.09	0.83	0.83	0.83	20.5	9	11.67	13.12
15	17	17	17	0.19	0.11	0.11	0.11	16.5	18	41.75	41.75
1	0	0	0	0.54	0.76	0.8	0.8	11.19	14.12	16.96	17.08
13	9	9	9	0.28	0.34	0.34	0.34	12.73	9.46	14.18	14.44
9	9	9	9	0.35	0.4	0.39	0.39	9.91	13.02	18.55	18.45
1	0	0	0	0.65	0.88	0.92	0.92	13.04	13.7	14.77	14.84
8	10	10	10	0.38	0.36	0.36	0.36	14.91	11.75	15.37	15.71
5	0	0	0	0.39	0.83	0.83	0.83	12.77	14.59	16.44	16.53
3	0	0	0	0.56	0.9	0.91	0.91	24.45	25.25	26.27	26.37
0	0	0	0	0.75	0.83	0.83	0.83	6.8	8	9	9
0	2	1	1	0.5	0.47	0.58	0.58	9.08	11	15.38	15.53

We then observed that our approach can preform better than this two traditional strategies, the advantages are shown in the more accurate MFS and less irrelevant schemas. Our approach also identified the least number of sub-schemas. As for the ‘parent-schema’ metric, our approach performs better than ‘distinguish faults’ but not as good as ‘regard as one fault’. This is because our strategy is actually based on the ‘distinguish faults’ strategy (the main difference is that our strategy filter these unsatisfied test cases). However, our approach is poor at the ‘ignored MFS’ metric, the possible reason for this may be that the unsatisfied test cases we discard may contain some useful information about the MFS. Above all, our approach get the best performance comparing with another two strategies, which can be shown in the ‘overall’ metric. This overall metric indicates that our approach performs better than ‘distinguish faults’ strategy, and which is better than the ‘regard as one fault’ strategy.

We further observed that, the extra cost (generated test cases) for our approach is acceptable, in fact, when comparing with ‘distinguish strategy’, our approach in average need just 3~4 more test cases, and when comparing with ‘regard as one fault’, our approach can even need smaller test cases in some cases(JFlex 1.4.1, JFlex 1.4.2, synthez 3 and synthez 6). This is because the extra test cases the FCI needed lies on what the MFS is, the difference at the identified MFS for FCI approaches will make their needed test cases differs greatly, so that the cost for replacing test cases in our approach may just have little influence on the final needed test cases.

Therefore, the answer we got for **Q2** is: Our approach get a better performance than two traditional strategies when handling masking effects in a acceptable extra cost.

6.3. Evaluating the ILP-based test case searching method

The third empirical study aims to evaluate the efficiency of the ILP-based test case searching component in our approach. To conduct this study, we implement a FCI approach which is also augmented with *replacing test cases* strategy, but the test case replacing process is by random.

6.3.1. Study setup. The setup of this case study based on the second case study, in fact, we use the same SUT model in TableXXI. Then we apply the newly randomly searching based FCI approach on identifying the MFS in these prepared SUT. In order to avoid the bias that come from the randomness, we repeat the newly approach 30 times to identify the MFS in each failing test case. We will compute the average additional test cases as well as other metrics listed in the precise section of the random-based approach.

6.3.2. Result and discussion. The evaluation of this random-based approach is also listed in Table XXII. Comparing with our ILP-based approach, we observed that there is little distinction between them at the metric: accurate schemas, parent-schemas, sub-schemas, ignore schemas, irrelevant schemas(the ILP-based approach performs a slightly better, e.g., for subject HSQL2cr8, ILP-based approach identified less sub ,parent and irrelevant schemas than random-based). This is because this two approaches is both *test case replacing* strategy, so when examine a schema, both of this two approaches may get the same result, although the test cases generated will be different. However, when considering the cost each approach will take, we can find the ILP-based approach performs better, which can reduces in average 1~ 2 test cases than random-based one.

In summary, the answer for **Q3** is: How to search a satisfied test case do have influence on the performance for our approach especially at the number of needed extra test cases, and the ILP based test cases can handle the masking effects at a relatively small cost than then random-based approach.

6.4. compare with Feedback driven combinatorial testing

The FDA-CIT [Yilmaz et al. 2013] approach can handles masking effects so that the covering array it generates can cover all the t-way schemas without being masked by the MFS. There is an integrated FCI approach in of FDA-CIT, of which this FCI approach has two versions, i.e., ternary-class and multiple-class. In this paper we use the multiple-class version as our compared approach, as it is claimed that the multiple-class version performs better than the former in [Yilmaz et al. 2013].

6.4.1. Study setup. As the FCI approach of FDA-CIT a post-analysis(classified tree) technique on given test cases, in this paper the same as [Yilmaz et al. 2013] we fed the FDA-CIT the covering array as the input. The covering array we generated are ranged from 2-4 way. The covering array generating method we used is [Cohen et al. 2003],

as it can be easily extended with constraints dealing and seeds injecting [Cohen et al. 2007b], which is needed in the FDA-CIT process. As different test cases will influence the result of characterization process, so we repeated generating 30 different 2-4 way covering array and fed them to FDA-CIT. Then we record the results of this approach, which consists of the metrics mentioned in the second case study.

Besides the FDA-CIT, we also applied our ILP-based approach on generated the covering array. In specific, for each failing test case in the covering array, we separately apply our approach to identify the MFS in that case. In fact, we can reduce much extra test cases if we utilize the other test cases in the covering array [Li et al. 2012]), but we didn't utilize the information for simplify the experiment. We then merge all the test cases that our approach needed for each failing test case in the covering array, we also merged other metrics listed in the second case study for each failing test cases.

As our approach generated different test cases from the FDA-CIT, we also let the multiple-class FCI approach of FDA-CIT to characterize the MFS using the test cases generated by our approach, so that we can get a more fair result when evaluating the FCI approach.

6.4.2. Result and discussion. We list the average result of the 30 times experiment for FDA-CIT, ILP-based approach, and FDA-CIT using our test cases (FDAs) respectively in Table XXIII. The result is organised almost the same way as Table XXII, except we added t column which indicates the strength of the covering array we generated for this experiment.

Table XXIII: Comparison with FDA-CIT

Subject	t	accurate			sub			parent			ignore		
		FDA	ILP	FDA-s	FDA	ILP	FDA-s	FDA	ILP	FDA-s	FDA	ILP	FDA-s
HSQL2cr8	2	0.17	2.27	1.57	0.57	0	0	0.17	0.4	2.17	3.87	2.3	2
	3	1.47	3.67	1	0	0	0	4.67	2	6.07	0.63	0.3	0.17
	4	0.83	4.8	1	0	0	0	9.03	3.37	8	0	0	0
HSQL2.2.5	2	1	1.97	0.37	0	0	0	2.4	0.73	3.8	0.4	0	0
	3	0	2	0.4	0	0	0	5	1	3.8	0	0	0
	4	0	2	0.33	0	0	0	5	1	4	0	0	0
HSQL2.2.9	2	0.9	1.77	0.9	0	0.77	9	1.47	0.47	6.8	1.93	0.53	0
	3	1	2	0.83	0	1	0	5.13	0.93	7.1	0.2	0	0
	4	1	2	1	0	1	0	5.87	1	6.7	0	0	0
JFlex 1.4.1	2	0	2	0	0	0	0	4.03	0	4	0	0	0
	3	0	2	0	0	0	0	4	0	0	0	0	4
	4	0	2	0	0	0	0	4	0	0	0	0	0
JFlex 1.4.2	2	0.3	1.97	0.93	0	0	0	3.6	1	2.16	0.03	0	0
	3	0	2	0.97	0	0	0	5	1	2.1	0	0	0
	4	0	2	1	0	0	0	5	1	2	0	0	0
synthez 1	2	0.97	1	1	0	0	0	1.7	1.93	2	0	0.07	0
	3	1	1	1	0	0	0	2	2	2	0	0	0
	4	1	1	1	0	0	0	2	2	2	0	0	0
synthez 2	2	0.17	1.3	0.73	0.37	0	0	0	0.4	2.37	2.27	1.2	1.03
	3	0.73	2.23	0.5	0	0	0	1.9	1.3	7.1	1.2	0.43	0.53
	4	0.63	2.97	0.1	0	0	0	5.3	2.33	16.1	0.53	0	0
synthez 3	2	0.43	2.97	0.73	0	0.93	0	4.3	1.73	5.3	0.47	0.17	0.5

Continued on next page

	3	0.2	3	0.87	0	1.57	0	7.2	3.67	6.57	0.07	0	0
	4	0.03	3	1	0	1.97	0	10.4	3	6	0	0	0
synthesez 4	2	0.3	2.3	0.33	0.07	0.63	0	2.63	1.97	7.7	1.93	0.63	0.4
	3	0.37	2.97	0.07	0	1.26	0	6.5	3.53	10.97	0.83	0.07	0
	4	0.07	3	0	0	1.77	0	11.7	4.67	11.4	0	0	0
synthesez 5	2	0.2	1.2	0.8	0.3	0	0	0.1	0.03	0.83	1.4	0.77	0.97
	3	0.87	1.4	0.53	0	0	0	0.5	0.23	3.03	1	0.6	0.77
	4	0.7	1.9	0.37	0	0	0	1.77	0.33	6.5	0.9	0.1	0.03
synthesez 6	2	0.23	2.63	0.17	0.2	2	0	2.93	1.63	9.63	2.6	0.5	0.4
	3	0.1	3	0.1	0	2.83	0	7.4	3.83	12.5	1.2	0.17	0.03
	4	0	3	0	0	3.8	0	10.2	6.03	14.5	0.47	0	0
synthesez 7	2	0.13	1.43	0.83	0.23	0	0	0.1	0.63	1.4	2.53	1.03	0.93
	3	0.87	2.17	0.93	0	0	0	0.43	1.23	2.97	1.77	0.17	0.13
	4	1	2.87	1	0	0	0	3.23	2.53	4.6	0.27	0	0
synthesez 8	2	0	0.2	0.17	0.03	0	0	0	0	0.03	0.3	0.13	0.13
	3	0	0.6	0.5	0.1	0	0	0	0	0.03	0.97	0.47	0.53
	4	0	1.33	0.8	0.1	0	0	0	0.07	0.67	1.53	0.4	0.5
synthesez 9	2	1	1	1	0	0	0	0.46	0.6	0.77	0.53	0	0.23
	3	1	1	1	0	0	0	1	1	1	0	0	0
	4	1	1	1	0	0	0	1	1	1	0	0	0
synthesez10	2	0	0.63	0	0.6	1	0.2	0	0	0.83	1.8	0.37	1.9
	3	0.07	0.97	0	0.23	1	0.03	0.36	0	1.9	2.23	0.03	1.97
	4	0	1	0	0.07	1	0	1.7	0	2	1.87	0	2

	irrelevant			overall			test cases		
t	FDA	ILP	FDA-s	FDA	ILP	FDA-s	FDA	ILP	FDA-s
2	2.53	0	1.97	0.12	0.51	0.39	23.6	70.1	70.1
3	3	0	1.47	0.51	0.87	0.6	76.6	241.8	241.8
4	0.97	0	0	0.65	0.9	0.71	183.5	606.6	606.6
2	1.4	0	0.1	0.38	0.87	0.56	26.7	68.8	68.8
3	0	0	0	0.52	0.83	0.56	67	202.4	202.4
4	0	0	0	0.53	0.83	0.56	130.1	503.3	503.3
2	2.37	0	0.2	0.28	0.72	0.58	29.2	78.3	78.3
3	0.1	0	0	0.61	0.8	0.61	72.8	221.7	221.7
4	0	0	0	0.64	0.8	0.62	129.8	560.3	560.3
2	0	0	0	0.49	1	0.5	30.5	87.3	87.3
3	0	0	0	0.5	1	0.5	73.4	269.2	269.2
4	0	0	0	0.5	1	0.5	190.6	724.7	724.7
2	0.63	0	0	0.5	0.83	0.62	34.3	106.9	106.9
3	0.03	0	0	0.52	0.83	0.61	72.3	305.7	305.7
4	0	0	0	0.53	0.83	0.61	186.8	836.9	836.9
2	0.33	14.3	0	0.66	0.13	0.78	40.3	342.87	342.87
3	0	16.73	0	0.78	0.12	0.78	93.4	809.1	809.1
4	0	17	0	0.78	0.12	0.78	218.8	1532.8	1532.8
2	1.37	0	1.2	0.11	0.52	0.4	19.77	54.4	54.4
3	2.2	0	1.33	0.36	0.82	0.52	59.5	171.5	171.5
4	2.6	0	1	0.44	0.89	0.54	152.7	415.1	415.1

Continued on next page

2	1.03	3.77	1.13	0.37	0.46	0.37	48.6	138.7	138.7
3	0.83	6.77	0.07	0.38	0.38	0.44	106.3	315.3	315.3
4	0.43	8.56	0	0.38	0.34	0.45	147.9	565.7	565.7
2	3.4	1.4	1.97	0.24	0.6	0.44	42.7	142.2	142.2
3	2.5	3.43	1.03	0.39	0.54	0.51	86.5	373.2	373.2
4	1.33	6.73	0.03	0.48	0.44	0.55	202.2	899.7	899.7
2	0.7	0	1	0.2	0.59	0.4	21.9	46.9	46.9
3	0.37	0	1.63	0.46	0.71	0.43	76.9	150.3	150.3
4	1.87	0	2.03	0.34	0.92	0.54	232.9	433.2	433.2
2	3.03	3.7	2	0.19	0.42	0.37	45.7	132.6	132.6
3	2.3	6.5	0.67	0.31	0.38	0.43	99.5	338.9	338.9
4	1.8	9.1	0.03	0.37	0.36	0.44	152.6	781.9	781.9
2	1.93	0	1.97	0.09	0.61	0.38	20.3	58.8	58.8
3	3.2	0	2.87	0.2	0.88	0.44	52.6	164.7	164.7
4	4.5	0	2.27	0.35	0.9	0.51	145.3	413.1	413.1
2	0.17	0	0.3	0.01	0.1	0.05	16.1	45.2	45.2
3	0.63	0	0.87	0.02	0.3	0.17	43.1	64.3	64.3
4	1.4	0	0.93	0.04	0.67	0.41	109.3	145.6	145.6
2	0.63	0.6	0.67	0.54	0.7	0.6	36.2	43.4	43.4
3	0	0	0	0.83	0.83	0.83	84.3	145	145
4	0	0	0	0.83	0.83	0.83	188	291.6	291.6
2	1.5	0.3	3.97	0.23	0.61	0.17	23.4	84.9	84.9
3	4.03	0.53	3.97	0.13	0.66	0.2	73.4	263.2	263.2
4	4.03	1	4	0.21	0.58	0.2	202.2	685.9	685.9

From this result, we can first observe that in all the cases our ILP-based approach can identify more accurate MFS and less ignored MFS than FDA-CIT approach. For the metric ‘parent-schema’, ‘sub-schema’, ‘irrelevant schemas’, there is ups and downs on both sides. With respect to the ‘overall metric’, we can find our approach have a significant advantage over the FDA-CIT, but it also needed much more test cases than FDA-CIT.

It is noted that, when apply multiple-class FCI on the test cases generated using our approach, their ‘overall’ metric is still not as good as our ILP-based approach, but may get some improvement than the original FDA-CIT.

Another interesting observation is that overall performance in most cases is increasing with the t , it can be easily understood: more test cases will contain more information about the MFS, so that we can utilize them to identified the MFS more precisely.

So the answer for the **Q4** is: Our approach can get a more precise result about the MFS, and the FDA-CIT can make the identifying process using small amount extra test cases. Both of this two approaches have their ups and downs, choosing which approach in practice is depending on the specific scenario you test.

6.5. Threats to validity

There are several threats to validity for these empirical studies. First, we have only surveyed two open-source software with five different versions, of which the program scale is medium-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 threat is that we just evaluated one FCI approach – FIC_BS [Zhang and Zhang 2011], further works needed to examine more algorithms in this field to get a more general result.

7. RELATED WORKS

Shi and Nie presented a further testing strategy for fault revealing and failure diagnosis [Shi et al. 2005], which first tests SUT with a covering array, then reduces the value schemas contained in the failed test case by eliminating those appearing in the passed test cases. If the failure-causing schema is found in the reduced schema set, failure diagnosis is completed with the identification of the specific input values which caused the failure; otherwise, a further test suite based on SOFOT is developed for each failed test case, testing is repeated, and the schema set is then further reduced, until no more failure is found or the fault has been located. Based on this work, Wang proposed an AIFL approach which extended the SOFOT process by mutating the changing strength in each iteration of characterizing failure-inducing combinations [Wang et al. 2010].

Nie et al. introduced the notion of Minimal Failure-causing Schema (MFS) and proposed the OFOT approach which extended from SOFOT that can isolate the MFS in SUT [Nie and Leung 2011a]. The approach mutates one value with different values for that parameter, hence generating a group of additional test cases each time to be executed. Compared with SOFOT, this approach strengthens the validation of the factor under analysis and also can detect the newly imported faulty combinations.

Delta debugging [Zeller and Hildebrandt 2002] proposed by Zeller is an adaptive divide-and-conquer approach to locate interaction fault. It is very efficient and has been applied to real software environment. Zhang et al. also proposed a similar approach that can identify the failure-inducing combinations that has no overlapped part efficiently [Zhang and Zhang 2011]. Later Li improved the delta-debugging based failure-inducing combination by exploiting the useful information in the executed covering array [Li et al. 2012].

Colbourn and McClary proposed a non-adaptive method [Colbourn and McClary 2008]. Their approach extends the covering array to the locating array to detect and locate interaction faults. C. Martinez 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 [Martínez et al. 2008; 2009]. Their algorithms focus on identifying the faulty tuples that have no more than 2 parameters.

Ghandehari et al. defined the suspiciousness of tuple and suspiciousness of the environment of a tuple [Ghandehari et al. 2012]. Based on this, they rank the possible tuples and generate the test configurations. They further utilized the test cases generated from the inducing combination to locate the faults inside the source code [Ghandehari et al. 2013].

Yilmaz proposed a machine learning method to identify inducing combinations from a combinatorial testing set [Yilmaz et al. 2006]. They construct a classified tree to analyze the covering arrays and detect potential faulty combinations. Beside this, Fouché [Fouché et al. 2009] and Shakya [Shakya et al. 2012] made some improvements in identifying failure-inducing combinations based on Yilmaz's work.

Our previous work [Niu et al. 2013] have proposed an approach that utilize the tuple relationship tree to isolate the failure-inducing combinations in a failing test case. One

novelty of this approach is that it can identify the overlapped faulty combinations. This work also alleviates the problem of introducing newly failure-inducing combinations in additional test cases.

In addition to the works that aims at identifying the failure-inducing combinations in test cases, there are some studies focus on working around the masking effects:

With having known masking effects in prior, Cohen [Cohen et al. 2007a; 2007b; 2008] studied the impacts that the masking effects render some generated test cases invalid in CT, and they proposed the approach that integrate the incremental SAT solver with covering array generating algorithms to avoid these masking effects in test cases generating process. Further study was conducted [Petke et al. 2013] to show the fact that with considering constraints, the higher-strength covering arrays with early fault detection is practical. Besides, additional constraints impacts in CT were studied in works like [Garvin et al. 2011; Bryce and Colbourn 2006; Calvagna and Gargantini 2008; Grindal et al. 2006; Yilmaz 2013].

Chen et al. addressed the issues of shielding parameters in combinatorial testing and proposed the Mixed Covering Array with Shielding Parameters (MCAS) to solve the problem caused by shielding parameters [Chen et al. 2010]. The shielding parameters can disable some parameter values to expose additional interaction errors, which can be regarded as a special case of masking effects.

Dumlu and Yilmaz proposed a feedback-driven approach to work around the masking effects [Dumlu et al. 2011]. In specific, it first use CTA classify the possible failure-inducing combinations and then eliminate them and generate new test cases to detect possible masked interaction in the next iteration. They further extended their work [Yilmaz et al. 2013], in which they proposed a multiple-class CTA approach to distinguish faults in SUT. In addition, they empirically studied the impacts on both ternary-class and multiple-class CTA approaches.

Our work differs from these ones mainly in the fact that we formally studied the masking effects on FCI approaches and further proposed a divide-and-conquer strategy to alleviate this impact.

8. CONCLUSIONS

Masking effects of multiple faults in SUT can bias the result of traditional failure-inducing combinations identifying approaches. In this paper, we formally analysed the impact of masking effects on FCI approaches and showed that both two traditional strategies are inefficient in handling such impact. We further presented a divide and conquer strategy for FCI approaches to alleviate this impact.

In the empirical studies, we extended FCI approach – FIC_BS [Zhang and Zhang 2011] with our strategy. The comparison between our approach with its traditional ones were conducted on several open-source software. The results indicated that our strategy do assist traditional FCI approach in getting a better performance when facing masking effects in SUT. We also empirically evaluated the efficiency of the test case searching component in our approach by comparing it with randomly searching based FCI approach, of which the result shows that the ILP based test case searching method can perform much more efficient. At last, we compared our approach with existed masking effects handling technique – FDA-CIT [Yilmaz et al. 2013], and observed that our approach can get a more precise result which can support better debugging aids, though our approach need generating more test cases than FDA-CIT.

As a future work, we need to do more empirical studies to make our conclusion more general. Our current experimental subjects are several middle-sized software, we would like to extend our approach into more complicated and large-scaled testing scenarios. Another promising work in the future is to combine white-box testing technique to make the FCI approaches get more accurate results when handling masking

effects. We believe that figuring out the fault levels of different bugs through white-box testing technique is helpful to reduce misjudgement in the failure-inducing combinations identifying process. At last, as the extent to what the FCI suffers from masking effects varies in different algorithms, the combination of different FCI approaches is desired in the future to further improve the performance for identifying MFS for multiple faults.

REFERENCES

- James Bach and Patrick Schroeder. 2004. Pairwise testing: A best practice that isn't. In *Proceedings of 22nd Pacific Northwest Software Quality Conference*. Citeseer, 180–196.
- Michel Berkelaar, Kjell Eikland, and Peter Notebaert. 2004. lp_solve 5.5, Open source (Mixed-Integer) Linear Programming system. Software. (May 1 2004). <http://lpsolve.sourceforge.net/5.5/> Last accessed Dec, 18 2009.
- Renée C Bryce and Charles J Colbourn. 2006. Prioritized interaction testing for pair-wise coverage with seeding and constraints. *Information and Software Technology* 48, 10 (2006), 960–970.
- Renée C Bryce, Charles J Colbourn, and Myra B Cohen. 2005. A framework of greedy methods for constructing interaction test suites. In *Proceedings of the 27th international conference on Software engineering*. ACM, 146–155.
- Andrea Calvagna and Angelo Gargantini. 2008. A logic-based approach to combinatorial testing with constraints. In *Tests and proofs*. Springer, 66–83.
- Baiqiang Chen, Jun Yan, and Jian Zhang. 2010. Combinatorial testing with shielding parameters. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*. IEEE, 280–289.
- David M. Cohen, Siddhartha R. Dalal, Michael L Fredman, and Gardner C. Patton. 1997. The AETG system: An approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on* 23, 7 (1997), 437–444.
- Myra B Cohen, Charles J Colbourn, and Alan CH Ling. 2003. Augmenting simulated annealing to build interaction test suites. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. IEEE, 394–405.
- Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2007a. Exploiting constraint solving history to construct interaction test suites. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007*. IEEE, 121–132.
- Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2007b. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 129–139.
- Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2008. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *Software Engineering, IEEE Transactions on* 34, 5 (2008), 633–650.
- Myra B Cohen, Peter B Gibbons, Warwick B Mugridge, and Charles J Colbourn. 2003. Constructing test suites for interaction testing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 38–48.
- Charles J Colbourn and Daniel W McClary. 2008. Locating and detecting arrays for interaction faults. *Journal of combinatorial optimization* 15, 1 (2008), 17–48.
- Emine Dumlu, Cemal Yilmaz, Myra B Cohen, and Adam Porter. 2011. Feedback driven adaptive combinatorial testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 243–253.
- Sandro Fouché, Myra B Cohen, and Adam Porter. 2009. Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 177–188.
- Brady J Garvin, Myra B Cohen, and Matthew B Dwyer. 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering* 16, 1 (2011), 61–102.
- Laleh Sh Ghandehari, Yu Lei, David Kung, Raghu Kacker, and Richard Kuhn. 2013. Fault localization based on failure-inducing combinations. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 168–177.
- Laleh Shikh Gholamhossein Ghandehari, Yu Lei, Tao Xie, Richard Kuhn, and Raghu Kacker. 2012. Identifying failure-inducing combinations in a combinatorial test set. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 370–379.

- Mats Grindal, Jeff Offutt, and Jonas Mellin. 2006. Handling constraints in the input space when using combination strategies for software testing. (2006).
- Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA Data Mining Software: An Update; SIGKDD Explorations, Volume 11, Issue 1. (2009).
- Jie Li, Changhai Nie, and Yu Lei. 2012. Improved Delta Debugging Based on Combinatorial Testing. In *Quality Software (QSIC), 2012 12th International Conference on*. IEEE, 102–105.
- Conrado Martínez, Lucia Moura, Daniel Panario, and Brett Stevens. 2008. Algorithms to locate errors using covering arrays. In *LATIN 2008: Theoretical Informatics*. Springer, 504–519.
- Conrado Martínez, Lucia Moura, Daniel Panario, and Brett Stevens. 2009. Locating errors using ELAs, covering arrays, and adaptive testing algorithms. *SIAM Journal on Discrete Mathematics* 23, 4 (2009), 1776–1799.
- Changhai Nie and Hareton Leung. 2011a. The minimal failure-causing schema of combinatorial testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20, 4 (2011), 15.
- Changhai Nie and Hareton Leung. 2011b. A survey of combinatorial testing. *ACM Computing Surveys (C-SUR)* 43, 2 (2011), 11.
- Xintao Niu, Changhai Nie, Yu Lei, and Alvin TS Chan. 2013. Identifying Failure-Inducing Combinations Using Tuple Relationship. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 271–280.
- Justyna Petke, Shin Yoo, Myra B Cohen, and Mark Harman. 2013. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 26–36.
- Kiran Shakya, Tao Xie, Nuo Li, Yu Lei, Raghu Kacker, and Richard Kuhn. 2012. Isolating Failure-Inducing Combinations in Combinatorial Testing using Test Augmentation and Classification. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 620–623.
- Liang Shi, Changhai Nie, and Baowen Xu. 2005. A software debugging method based on pairwise testing. In *Computational Science-ICCS 2005*. Springer, 1088–1091.
- Charles Song, Adam Porter, and Jeffrey S Foster. 2012. iTree: Efficiently discovering high-coverage configurations using interaction trees. In *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 903–913.
- Ziyuan Wang, Baowen Xu, Lin Chen, and Lei Xu. 2010. Adaptive interaction fault location based on combinatorial testing. In *Quality Software (QSIC), 2010 10th International Conference on*. IEEE, 495–502.
- Cemal Yilmaz. 2013. Test case-aware combinatorial interaction testing. *Software Engineering, IEEE Transactions on* 39, 5 (2013), 684–706.
- Cemal Yilmaz, Myra B Cohen, and Adam A Porter. 2006. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on* 32, 1 (2006), 20–34.
- Cemal Yilmaz, Emine Dumlu, M Cohen, and Adam Porter. 2013. Reducing Masking Effects in Combinatorial Interaction Testing: A Feedback Driven Adaptive Approach. (2013).
- Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on* 28, 2 (2002), 183–200.
- Zhiqiang Zhang and Jian Zhang. 2011. Characterizing failure-causing parameter interactions by adaptive testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 331–341.