Identifying minimal failure-causing schemas in the presence of multiple faults

XINTAO NIU and CHANGHAI NIE, State Key Laboratory for Novel Software Technology, Nanjing University
JEFF Y. LEI, Department of Computer Science and Engineering, The University of Texas at Arlington
Hareton Leung and ALVIN CHAN, Department of Computing, Hong Kong Polytechnic University
Xiaoyin Wang, Department of Computer Science, University of Texas at San Antonio

Combinatorial testing (CT) has been proven effective in revealing the failures caused by the interaction of factors that affect the behavior of a system. The theory of Minimal Failure-Causing Schema (MFS) has been proposed to isolate the root cause of a failure after CT. Most algorithms that aim to identify MFS focus on handling a single fault in the System Under Test (SUT). However, we argue that multiple faults are more common in practice, under which masking effects may be triggered so that some failures cannot be observed. The traditional MFS theory lacks a mechanism to handle such effects; hence, they may incorrectly isolate the MFS. To address this problem, we propose a new MFS model that takes into account multiple faults. We first formally analyse the impact of the multiple faults on existing MFS identifying algorithms, especially in situations where masking effects are triggered by multiple faults. We then develop an approach that can assist traditional algorithms to better handle multiple faults. Empirical studies were conducted using several kinds of open-source software, which showed that multiple faults with masking effects do negatively affect traditional MFS identifying approaches and that our approach can help 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, Jeff Y. Lei, Hareton Leung, and Alvin Chan, 2015.Identifying minimal failure-causing schemas in the presence of multiple faults. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2010), 33 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 affect the behaviour of the SUT. The failures caused by the interaction of these factors can make software testing challenging, especially when the interaction space is large. In the worst case, we need to examine every possible interaction of these factors as each interaction may cause unique

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)

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	On	PASS
2	Off	Off	On	On	PASS
3	Off	On	Off	Off	Fail
4	On	Off	Off	On	PASS
5	On	Off	On	Off	PASS

failure [Song et al. 2012]. While exhaustive testing achieves maximal test coverage, it is impractical and uneconomical. One remedy for this problem is Combinatorial Testing (CT)[1], which systematically samples the interaction space and selects a relatively small set of test cases that cover all valid interactions, with the number of factors involved in each interaction no more than a prior fixed integer, i.e., the *strength* of the interaction. Many works in CT aim to construct the smallest set of test cases [Cohen et al. 1997; Bryce et al. 2005; Cohen et al. 2003; Lei et al. 2008], which is also called *covering array*.

Once failures are detected by a covering array, the failure-inducing interactions in the failing test cases should be isolated. This task is important as it can facilitate debugging efforts by reducing the scope of code that is needed for inspection. [Ghandehari et al. 2012]. However, information from a covering array sometimes is not sufficient to identify the failure-inducing interactions [Colbourn and McClary 2008]. Thus, additional information is often needed. Consider the following example [Bach and Schroeder 2004], Table I presents a two-way covering array for testing an MS-Word application in which we want to examine various interactions of options for 'Highlight', 'Status Bar', 'Bookmarks' and 'Smart tags'. Assume the third test case failed. We can get six two-way suspicious interactions that may be responsible for this failure. They are (Highlight: Off, Status Bar: On), (Highlight: Off, Bookmarks: Off), (Highlight: Off, Smart tags: Off), (Status Bar: On, Bookmarks: Off), (Status Bar: On, Smart tags: Off), and (Bookmarks: Off, Smart tags: Off). Without additional information, it is difficult to figure out the specific interactions in this suspicious set that caused the failure. In fact, considering that the higher strength interactions could also be failure-inducing interactions, e.g., (Highlight: Off, Status Bar: On, Smart tags: Off), the problem becomes more complicated.

To address this problem, prior work [Nie and Leung 2011a] specifically studied the properties of the failure-inducing interactions in the SUT, based on which additional test cases were generated to identify them. Other approaches to identify the failure-inducing interactions in the SUT include building a tree model [Yilmaz et al. 2006], adaptively generating additional test cases according to the outcome of the last test case [Zhang and Zhang 2011], ranking suspicious interactions based on some rules [Ghandehari et al. 2012], and using graphic-based deduction [Martínez et al. 2008], among others.

Most existing approaches mainly focus on the ideal scenario in which SUT only contains one fault, under which the outcomes of test cases can be simply categorized into failure or pass. However, in this paper, we argue that SUT with multiple faults is the more common testing scenario in practice, which will result in many distinguished failures as outcomes of test cases, and moreover, this affects the effectiveness of Failure-inducing Interactions Identifying (FII) approaches. The major challenge imposed by multiple faults is dealing with the masking effect. A masking effect [Dumlu et al. 2011; Yilmaz et al. 2014] occurs when some failures prevent test cases from checking interactions that are supposed to be tested. Take the Linux command *Grep* for exam-

[1]Another term for CT is Combinatorial Interaction Testing, which is abbreviated as CIT. In this paper, they are uniformly cited as Combinatorial testing (CT).

ple. We notice that there are two different failures reported in the bug tracker system. The first [2] claims that Grep incorrectly matches unicode patterns with '\<\>', while the second [3] claims an incompatibility between option '-c' and '-o'. When we put these two scenarios into one test case, only one failure will be observed, which means the other failure is masked by the observed failure. This effect prevents test cases from being executed normally, leading to incorrect judgment of the correlation between the interactions checked in the test case and the failure that has been masked. This effect was firstly noted by Dumlu and Yilmaz in [Dumlu et al. 2011], in which they found that the masking effect in CT can prevent traditional covering array in testing some interactions.

As masking effect can negatively affect the performance of FII approaches, a natural question is how this effect biases the results of these approaches. In this paper, we formalize the process of identifying the failure-inducing interactions under the circumstances in which masking effects exist in the SUT and try to answer this question. One insight from the formal analysis is that we cannot completely avoid the impact of masking effects even if we do exhaustive testing. Even worse, either ignoring the masking effects or treating different failures as the same failure is detrimental to the FII process.

To address this concern, we propose a strategy to alleviate this impact by adopting a divide and conquer framework. With this framework, FII approaches are scheduled to separately handle each failure in the SUT. Specifically, for a particular failure, FII approaches only focus on the test cases that either pass or trigger the same failure under analysis. Test cases that triggered other different failures will be replaced with some newly generated test cases. In this way, FII approaches can properly work with little interference from the negative masking effects.

To evaluate the effectiveness of our approach, we applied our strategy on the FII approach FIC_BS [Zhang and Zhang 2011]. The subjects used were seven open-source software systems found in the developers' forum. Through studying their bug reports in the bug tracker system as well as their user manuals, we built a testing model which can reproduce the reported bugs with given test cases. We then compared the FII approach augmented with our strategy to the original FII approach. We further compared our approach with the only existing masking handling technique – FDA-CIT [Yilmaz et al. 2014]. Our studies show that our replacing strategy achieved a better performance than the traditional approaches when the subject encountered multiple faults, especially when these faults can induce masking effects.

The main contributions of this paper are:

— We formally analysed the relationships between failure-inducing interactions and test sets. (Section 3)
— We studied the impact of the masking effects caused by multiple faults on FII approaches. (Section 4)
— We proposed an efficient test case replacement strategy to alleviate the impact of these effects, and showed an MFS identification approach augmented with this replacement strategy (Section 5).
— We conducted several empirical studies and showed that our strategy can assist FII approaches to achieve better performance in identifying failure-inducing interactions in the SUT with masking effects. (Section 7)

```
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 simple program *foo* with four input parameters

2. MOTIVATING EXAMPLE

We use a small example to motivate our approach. Assume a method *foo* has four input parameters: *a, b, c,* and *d*. The four parameter types are all integers and their values are: $v_a = \{7, 11\}, v_b = \{2, 4, 5\}, v_c = \{4, 6\}, v_d = \{3, 5\}$. The code of *foo* is shown in Figure 1.

There are two potential failures of *foo*: first, in step 1 we can get an *Arithmetic Exception* when b is equal to c, i.e., $b = 4$ and $c = 4$, that causes a division by zero. Second, another *Arithmetic Exception* will be triggered in step 2 when $c < d$, i.e., $c = 4$ and $d = 5$, taking square root of a negative number. So the expected failure-inducing interactions in this example should be (-, 4, 4, -) and (-, -, 4, 5).

FII approaches normally applies black-box testing, i.e., feed inputs to the programs and execute them to observe the result. The basic proposition of FII approaches is that the failure-inducing interactions for a particular failure can only appear in those test cases that trigger this failure. FII approaches often aim at using as few test cases as possible to get the same (or approximate) result as exhaustive testing, so the results derived from an exhaustive testing set are the best that FII approaches can achieve. Here, we will show how exhaustive testing works to identify the failure-inducing interactions for the program.

Table II. Test cases and their corresponding result

id	test case	result	id	test case	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

We first generate every possible test case listed in the column "test case" of Table II. The execution results are listed in the result column of Table II. In this column, *PASS* means that the program runs without any exception. *Ex 1* indicates that the program triggered an exception corresponding to step 1 and *Ex 2* indicates the program

[2]http://savannah.gnu.org/bugs/?29537
[3]http://savannah.gnu.org/bugs/?33080

triggered an exception corresponding to step 2. From the data listed in Table II, we can determine that (-, 4, 4, -) must be the failure-inducing interaction of Ex 1 as all the test cases that triggered Ex 1 contain this interaction. Similarly, interactions (-, 2, 4, 5) and (-, 5, 4, 5) must be the failure-inducing interactions of Ex 2. We list these interactions and the corresponding exceptions in Table III.

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

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

Note that in this example we did not get the expected result with traditional FII approaches. The failure-inducing interactions for Ex 2 are (-,2,4,5) and (-,5,4,5), respectively, instead of the expected interaction (-,-,4,5). So why did we fail to get the (-,-,4,5)? The reason lies in *test case 6* (7,4,4,5) and *test case 18* (11,4,4,5). These two test cases contain the interaction (-,-,4,5), but they did not trigger Ex 2; instead, Ex 1 was triggered.

Getting back to the program *foo*, it can be found that if Ex 1 is triggered, it will stop executing the remaining code and report the exception result. In other words, Ex 1 may mask Ex 2. Let us re-examine the interaction (-,-,4,5). If we suppose that *test case 6* and *test case 18* should trigger Ex 2 if they did not trigger Ex 1, then we can conclude that (-,-,4,5) should be the failure-inducing interaction of Ex 2, which is identical to the expected one. However, we cannot get this result, unless we fix the code that triggers Ex 1 and re-execute all the test cases.

So in practice, *when we lack resources to execute all the test cases repeatedly or can only do black-box testing, a more economical and efficient approach to alleviate the masking effect on FII approaches is desired*.

3. FORMAL MODEL OF MINIMAL FAILURE-CAUSING SCHEMA

This section presents some definitions and propositions for a formal description of failure-inducing interactions and test sets.

3.1. Minimal Failure-causing Schemas

Assume that the behaviour of a 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,...,k). In practice, these parameters can represent many factors, such as input variables, run-time options, building options, etc. Next we will give some formal definitions, Definitions 3.1, 3.3, 3.4 were originally defined in [Nie and Leung 2011b].

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

For the example in Section 2, ($a = 7$, $b = 2$, $c = 4$, $d = 3$) is a test case, which is actually a group of values being assigned to each input parameter.

Definition 3.2. A *failure* is an abnormal execution of a test case.

In CT, such a *failure* can be a thrown exception, compilation error, assertion failure or constraint violation. To facilitate our discussion, we introduce the following assumptions that will be used throughout this paper:

ASSUMPTION 1. *The execution result of a test case is deterministic.*

This assumption is a common assumption of CT fault diagnosis[Wang et al. 2010; Zhang and Zhang 2011; Ghandehari et al. 2012; Li et al. 2012; Niu et al. 2013]. It indicates that the outcome of executing a test case is reproducible and will not be affected by some random events.

ASSUMPTION 2. *Different failures in the SUT can be distinguished by various information such as exception traces, state conditions, or the like.*

This assumption indicates that the testers can detect different failures during testing. As different failures will complicate fault diagnosis tasks, distinguishing them is the first step to resolve them.

In this paper, failures are classified according to the specific *fault* information. For example, if failures have the same exception traces information, they are treated as the same failure. For convenience, when we refer to a failure in the remaining part of this paper, we mean *one type of failure* (with same fault information). Actually in many cases, the same type of failures is caused by the same fault.

In practice, these assumptions may not be satisfied. In Section 6, we will discuss their impact on the theories and approach proposed in this paper, as well as the measures to alleviate them. Now let us consider the condition that some failures are triggered by some test cases. It is then desirable to determine the cause of these failures and hence some parameter values of the failing test cases must be analysed.

Definition 3.3. For the SUT, the τ-tuple (-,v_{b_1},...,v_{b_τ},...) is called a τ-degree *schema* ($0 < \tau \leq k$) when some τ parameters have fixed values and the others can take on their respective allowable values, represented as "-".

When $\tau = k$, τ-degree *schema* is actually a test case. Furthermore, if every fixed value in a schema is in a test case, we say this test case *contains* the schema.

For example, (-, 4, 4, -) in Table III is a 2-degree schema. And the test case (7, 4, 4, 3) contains this schema.

Definition 3.4. Let c_1 be a *m*-degree schema, c_2 be an *n*-degree schema in the SUT, and $m < n$. If all the fixed parameter values in c_1 are also in c_2, then c_2 *subsumes* c_1. In this case, we can also say that c_1 is a *sub-schema* of c_2, and c_2 is a *super-schema* of c_1, denoted as $c_1 \prec c_2$.

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

According to definition 3.4, it is obvious that the subsuming relationship '$\prec$' is transitive, i.e., if $c_1 \prec c_2, c_2 \prec c_3$, then $c_1 \prec c_3$.

Definition 3.5. If for any test case, as long as it contains the schema c, it will trigger the particular failure F. Then we call c the *failure-causing schema* of F. Additionally, if none of the sub-schema of c is the *failure-causing schema* of F, we then call c the *Minimal Failure-causing Schema* (MFS) of F.

In fact, MFS is identical to the failure-inducing interactions discussed previously. Identifying MFS helps to focus on the root cause of a failure and thus facilitates the debugging efforts. Note that all the failures discussed in this paper are option-related [Yilmaz et al. 2006]. That is, all the failures are caused by the interactions of the parameter values in the SUT. Another noteworthy point is that, in practice, we may not accurately obtain the MFS according to the MFS definition (See Section 4). This is because in some specific cases, e.g., the masking problems which we will discuss later, we may not determine whether some test cases will trigger the particular failure of F, or not.

Some notations used later are listed below for ease of reference:

— k : The number of parameters that influence the SUT.
— V_i : The set of discrete values that the ith parameter of the SUT can take.
— T^* : The exhaustive set of test cases for the SUT. For a SUT with k parameters, and each parameter can take $|V_i|$ values, the number of test cases in T^* is $\prod_{i=1}^{k} |V_i|$. Note that some test cases may be invalid if there exists constraints among the parameters.
— T : A set of test cases. (Similarly for $T_i,\ T_j, \ldots.$)
— $\bar{T}$: The complementary test set of T, i.e., $\bar{T} \bigcup T = T^*,\ \bar{T} \bigcap T = \varnothing$.
— $A \backslash B$: The set of elements that belong to set A but not to B. For example $T_i \backslash T_j$ indicates the set of test cases that belong to set T_i, but not to T_j.
— L : The number of types of failures can be triggered in the SUT.
— F_m : The mth type of failure in the SUT ($1 \leq m \leq L$);
— T_{F_m} : All the test cases that can trigger the failure F_m in the SUT.
— $\mathcal{T}(c)$: All the test cases that contain the schema c in the SUT. Based on the definition of MFS, we know that if schema c is an MFS of F_m, then $\mathcal{T}(c) \subseteq T_{F_m}$. Note that, there may be multiple MFS for F_m
— $\mathcal{I}(t)$: All the schemas that are contained 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 contained in test set T, i.e., $\mathcal{I}(T) = \bigcup_{t \in T} \mathcal{I}(t)$.
— $\mathcal{S}(T)$: All the schemas that are only contained in test set T (Referred to as *Special schemas*); $\mathcal{S}(T) = \mathcal{I}(T) \backslash \mathcal{I}(\bar{T})$.
— $\mathcal{C}(T)$: A set of the minimal schemas that are only contained in test set T (Referred to as *Minimal schemas*); $\mathcal{C}(T) = \{c | c \in \mathcal{S}(T)\ and\ \nexists c' \prec c, s.t., c' \in \mathcal{S}(T)\}$.

3.2. Relations between schemas and test sets

PROPOSITION 3.6 (SMALLER SCHEMA c HAS A LARGER $\mathcal{T}(c)$). *For schemas c_1, c_2, if $c_1 \prec c_2$, then all the test cases that contain c_2 must also contain c_1, i.e., $\mathcal{T}(c_2) \subseteq \mathcal{T}(c_1)$.*

We omit the proof of this proposition as it is quite obvious. Suppose a SUT with four binary parameters, which can be denoted as SUT(2^4). Table IV illustrates an example of the Proposition 3.6. The left column lists the schema c_2 = (0,0,-,-) as well as all the test cases in $\mathcal{T}(c_2)$, while the right column lists the schema c_1 = (0,-,-,-) and $\mathcal{T}(c_1)$. We can see that when $c_1 \prec c_2$, $\mathcal{T}(c_2) \subseteq \mathcal{T}(c_1)$.

Table IV. Example of Proposition 3.6

c_2
(0, 0, -, -)
$\mathcal{T}(c_2)$
(**0**, **0**, 0, 0)
(**0**, **0**, 0, 1)
(**0**, **0**, 1, 0)
(**0**, **0**, 1, 1)

c_1
(0, -, -, -)
$\mathcal{T}(c_1)$
(**0**, 0, 0, 0)
(**0**, 0, 0, 1)
(**0**, 0, 1, 0)
(**0**, 0, 1, 1)
(**0**, 1, 0, 0)
(**0**, 1, 0, 1)
(**0**, 1, 1, 0)
(**0**, 1, 1, 1)

PROPOSITION 3.7 (PROPERTY OF SPECIAL SCHEMA SET OF TEST SET T). *For any test set T of the SUT, $\bigcup_{c \in \mathcal{S}(T)} \mathcal{T}(c) = T$.*

PROOF. As $\mathcal{S}(T) = \mathcal{I}(T) \backslash \mathcal{I}(\bar{T})$, $\forall c \in \mathcal{S}(T),\ c \in \mathcal{I}(T)\ and\ c \notin \mathcal{I}(\bar{T})$. Then $\forall t \in \mathcal{T}(c), t\ contains\ c$, indicating that $t \in T$. Hence, $\mathcal{T}(c) \subseteq T$. Then $\bigcup_{c \in \mathcal{S}(T)} \mathcal{T}(c) \subseteq T$.

On the other hand, $\forall t \in T$, $\exists c' \in \mathcal{I}(t)$, such that $c' \notin \mathcal{I}(\bar{T})$ (at least it holds when $c' = t$). Hence, $c' \in \mathcal{S}(T)$. Obviously $t \in \mathcal{T}(c') \subseteq \bigcup_{c\in\mathcal{S}(T)} \mathcal{T}(c)$. Therefore, $T \subseteq \bigcup_{c\in\mathcal{S}(T)} \mathcal{T}(c)$. □

PROPOSITION 3.8 (PROPERTY OF MINIMAL SCHEMA SET OF TEST SET T). *For any test set T of the SUT, $\bigcup_{c\in\mathcal{C}(T)} \mathcal{T}(c) = T$.*

PROOF. As $\mathcal{C}(T) = \{c | c \in \mathcal{S}(T)\ and\ \nexists c' \prec c, s.t., c' \in \mathcal{S}(T)\}$, indicating that $\mathcal{C}(T) \subseteq \mathcal{S}(T)$. It is then obviously $\bigcup_{c\in\mathcal{C}(T)} \mathcal{T}(c) \subseteq \bigcup_{c\in\mathcal{S}(T)} \mathcal{T}(c)$. Hence, we just need to prove that $\bigcup_{c\in\mathcal{S}(T)} \mathcal{T}(c) \subseteq \bigcup_{c\in\mathcal{C}(T)} \mathcal{T}(c)$.

$\forall t \in \bigcup_{c\in\mathcal{S}(T)} \mathcal{T}(c)$, $\exists c \in \mathcal{S}(T),\ s.t., t \in \mathcal{T}(c)$. According to the definition of $\mathcal{C}(T)$, $\exists c' \in \mathcal{C}(T), s.t.,\ c' = c\ or\ c' \prec c$. Correspondingly $\mathcal{T}(c') = \mathcal{T}(c),\ or\ \mathcal{T}(c) \subseteq \mathcal{T}(c')$ by Proposition 3.6. Hence, $t \in \mathcal{T}(c') \subseteq \bigcup_{c\in\mathcal{C}(T)} \mathcal{T}(c)$.

Therefore, $\bigcup_{c\in\mathcal{C}(T)} \mathcal{T}(c) = \bigcup_{c\in\mathcal{S}(T)} \mathcal{T}(c) = T$. □

Table V gives an example of T^*, T, $\bar{T}$, $\mathcal{S}(T)$ and $\mathcal{C}(T)$ in SUT(2^3). We can find that all the schemas in $\mathcal{S}(T)$ and $\mathcal{C}(T)$ are only contained in test set T, and for any $t \in T$, it contains at least one schema in $\mathcal{S}(T)$ and $\mathcal{C}(T)$. Additionally, $\mathcal{C}(T)$ is a minimal schema set which filters those super schemas in $\mathcal{S}(T)$.

Table V. Example of the special and minimal schemas

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

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

According to the definition of $\mathcal{C}(T)$, one observation is $\mathcal{C}(T) \subseteq \mathcal{S}(T)$, and for any schema in $\mathcal{S}(T)$, it either belongs to $\mathcal{C}(T)$, or is the super schema of one element of $\mathcal{C}(T)$, i.e., $\forall c \in \mathcal{S}(T), \exists c' \in \mathcal{C}(T), s.t., c' = c,\ or\ c' \prec c$.

PROPOSITION 3.9 (MINIMAL SCHEMA OF THE SUBSET OF TEST SET T). *For any test set T and schema c of the SUT, if $\mathcal{T}(c) \subseteq T$, then $c \in \mathcal{S}(T)$.*

PROOF. Assume $c \notin \mathcal{S}(T)$, i.e., $c \notin \mathcal{I}(T)\backslash\mathcal{I}(\bar{T})$, then $c \in \mathcal{I}(\bar{T})$. It indicates that $\exists t \in \bar{T}, t \in \mathcal{T}(c)$, which contradicts that $\mathcal{T}(c) \subseteq T$. Therefore, $c \in \mathcal{S}(T)$. □

Table VI shows an example of this proposition for SUT(2^3). In this table, the test set $\mathcal{T}(c)$ of schema c is the subset of test set T. As a result, the special schema set $\mathcal{S}(T)$ of T contains this schema c = (0, 0, -).

Based on Proposition 3.9, as long as $\mathcal{T}(c) \subseteq T$ for any schema c and any test set T in the SUT, c either belongs to $C(T)$ or is the super-schema of some schema in $C(T)$. Considering a more general scenario, i.e., two test sets T_1, T_2 with $T_2 \subseteq T_1$, we then can easily obtain the relationship between $\mathcal{C}(T_1)$ and $\mathcal{C}(T_2)$ according to Proposition 3.9.

PROPOSITION 3.10 (MINIMAL SCHEMAS IN THE SMALLER TEST SET). *For T_1 and T_2 of the SUT with $T_2 \subseteq T_1$, $\forall c_2 \in \mathcal{C}(T_2)$, $\exists c_1 \in \mathcal{C}(T_1), s.t.,\ either\ c_1 = c_2\ or\ c_1 \prec c_2$.*

Table VI. Example of a minimal schema of the subset of a test set

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

PROOF. $\forall c_2 \in \mathcal{C}(T_2)$, $\mathcal{T}(c_2) \subseteq T_2 \subseteq T_1$. According to Proposition 3.9, $c_2 \in \mathcal{S}(T_1)$. By definitions of $\mathcal{S}(T)$ and $\mathcal{C}(T)$, $\exists c_1 \in \mathcal{C}(T_1), s.t., c_1 = c_2,\ or\ c_1 \prec c_2$. □

PROPOSITION 3.11 (MINIMAL SCHEMAS IN THE LARGER TEST SET). *For* T_1 *and* T_2 *of the SUT,* $T_2 \subseteq T_1$. *Then* $\forall c_1 \in \mathcal{C}(T_1)$, $\exists c_2 \in \mathcal{C}(T_2), s.t.,\ (1)\ c_1 = c_2$, *or* $(2)\ c_1 \prec c_2$, *or* $(3)\ \nexists c_2 \in \mathcal{C}(T_2), s.t., c_2 \prec c_1\ or\ c_2 = c_1,\ or\ c_1 \prec c_2$.

PROOF. Assume that $\exists c_1 \in \mathcal{C}(T_1)$, s.t., $\exists c_2 \in \mathcal{C}(T_2), c_2 \prec c_1$. According to Proposition 3.10, as $T_2 \subseteq T_1$, so $\forall c_2 \in \mathcal{C}(T_2)$, $\exists c_1' \in \mathcal{C}(T_1), s.t.,\ either\ c_1' = c_2\ or\ c_1' \prec c_2$. Combining them, we can get that $c_1' \prec c_1$. This is a obvious contradiction, as c_1 and c_1' are both minimal schemas in $\mathcal{C}(T_1)$.

Hence, apart from the impossible relationship $c_2 \prec c_1$, then $\forall c_1 \in \mathcal{C}(T_1)$, $\exists c_2 \in \mathcal{C}(T_2), s.t.,\ (1)\ c_1 = c_2$, *or* $(2)\ c_1 \prec c_2$, *or* $(3)\ \nexists c_2 \in \mathcal{C}(T_2), s.t., c_2 \prec c_1\ or\ c_2 = c_1,\ or\ c_1 \prec c_2$. □

We need to note the third case, i.e., $\nexists c_2 \in \mathcal{C}(T_2), s.t., c_1 \prec c_2\ or\ c_1 = c_2,\ or\ c_2 \prec c_1$. We refer to this case as c_1 is *irrelevant* to $\mathcal{C}(T_2)$. Furthermore, we can also say a schema is *irrelevant* to another schema if these two schemas are neither identical nor subsuming each other.

We illustrate these scenarios with examples in Table VII for SUT(2^3). There are two parts in this table, with each part showing two test sets: T_1 and T_2, which have $T_2 \subseteq T_1$. In the left part, the schemas in $\mathcal{C}(T_2)$: (0, 0, -) and (0, -, 0), both are the super-schemas of the one in $\mathcal{C}(T_1)$:(0,-,-). While in the right part, the schemas in $\mathcal{C}(T_2)$: (0, 0, -) and (0, -, 0) are both in $\mathcal{C}(T_1)$. Furthermore, one schema in $\mathcal{C}(T_1)$: (1,1,-) is irrelevant to $\mathcal{C}(T_2)$.

Table VII. Minimal schemas of two subsuming test sets

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

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

In summary, these propositions provide the foundation of MFS identification (Propositions 3.7 and 3.8), and more importantly, they clarify the relationships between the minimal schemas of two different test sets (Propositions 3.11 and 3.10). As will be shown in Section 4, under the masking effects, the actual failing test set is either to be the super-set or sub-set of the observed failing test set. Hence, Propositions 3.11 and 3.10 can help to describe the impact of masking effects on MFS identification later.

4. MASKING EFFECT

As discussed before, $\mathcal{C}(T_{F_m})$ is considered to be the MFS set of failure F_m in theory. When accounting for the masking effects between multiple faults, however, this consideration is not correct.

Definition 4.1. A *masking effect* occurs when a test case t contains an MFS of a particular failure, but it does not trigger the expected failure because other unexpected event, such as other type of failure or unaccounted control dependency, was triggered earlier that prevents t from being normally checked.

Taking the masking effects into account, when identifying the MFS of a specific failure F_m, we must not ignore those test cases which did not trigger F_m but should have triggered it. We call these test cases $T_{mask(F_m)}$. Hence, the MFS of failure F_m should be revised as $\mathcal{C}(T_{F_m} \bigcup T_{mask(F_m)})$.

Going back to the motivating example in Section 2, as *test case 6* and *test case 18* should trigger Ex 2 in case they did not trigger Ex 1, $T_{mask(F_2)}$ is { (7,4,4,5),(11,4,4,5)}. Hence, the MFS of $Ex2$ is $\mathcal{C}(T_{F_2} \bigcup T_{mask(F_2)})$, which is (-,-,4,5) instead of the incorrect schema set $\{(-, 2, 4, 5), (-, 5, 4, 5)\}$.

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 some test cases to figure out $T_{mask(F_m)}$.

For traditional FII approaches, without the knowledge of $T_{mask(F_m)}$, two common strategies can be adopted to deal with the multiple faults problem, i.e., *regarded as same failure* and *distinguishing failures*. The former strategy treats all types of failures as the same failure, while the latter distinguishes the failures but with no special consideration of the masking effects, i.e., if a test case fails with a particular type of failure, this strategy presumes it does not contain other types of failures.

4.1. Regarded as same failure strategy

This is the most common strategy. With this strategy, the minimal schemas are the set $\mathcal{C}(\bigcup_{i=1}^{L} T_{F_i})$, where L is the number of types of failures in the SUT. Obviously, $T_{F_m} \bigcup T_{mask(F_m)} \subseteq \bigcup_{i=1}^{L} T_{F_i}$. By Proposition 3.11, some schemas obtained by this strategy may be the sub-schemas of some of the actual MFS, or be irrelevant to the actual MFS.

As an example, consider the test cases in Table VIII. Assume we need to characterize the MFS of *failure 1*. All the test cases that triggered *failure 1* are listed in column T_{F_1}; similarly, we list the test cases that triggered other types of failures in column $T_{mask(F_1)}$ and $T_{non\_mask(F_1)}$, respectively, in which the former masked *failure 1*, while the latter did not. Actually the MFS of *failure 1* should be (1,1,-,-) and (-,1,1,1) as listed in the column 'actual MFS of *failure 1*'. However, when we use the *regarded as same failure* strategy, the minimal schemas obtained will be (-,-,0,0), (1,1,-,-), (-,-,1,1), in which (-,-,0,0) is irrelevant to the actual MFS of *failure 1*, and (-,-,1,1) is a sub-schema of the actual MFS (-,1,1,1).

4.2. Distinguishing failures strategy

Distinguishing the failures by the exception traces or error code can help identify the MFS related to a particular failure. Yilmaz [Yilmaz et al. 2014] proposed the *multiple-class* failure characterizing method instead of the *ternary-class* approach to make the characterizing process more accurate. Besides, other approaches can also be easily extended using this strategy for testing the SUT with multiple faults.

This strategy focuses on identifying the set of $\mathcal{C}(T_{F_m})$. As $T_{F_m} \bigcup T_{mask(F_m)} \supseteq T_{F_m}$, by Proposition 3.10, some schemas obtained by this strategy may be the super-schema

Table VIII. Masking effects for exhaustive testing

T_{F_1}	$T_{mask(F_1)}$	$T_{non\_mask(F_1)}$
(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 of *Failure 1*	regarded as same failure	distinguishing failures
$\mathcal{C}(T_{F_1} \bigcup T_{mask(F_1)})$	$\mathcal{C}(T_{F_1} \bigcup T_{mask(F_1)} \bigcup T_{non\_mask(F_1)})$	$\mathcal{C}(T_{F_1})$
(1, 1, -, -)	(-, -, 0, 0)	(1, 1, -, 1)
(-, 1, 1, 1)	(1, 1, -, -)	(1, 1, 1, -)
	(-, -, 1, 1)	

of some actual MFS. Moreover, some actual MFS may be irrelevant to the schemas obtained by this strategy according to Proposition 3.11, which means that this strategy will *ignore* these actual MFS.

For the simple example shown in Table VIII, when using this strategy, we will get the minimal schemas (1, 1, -, 1) and (1, 1, 1, -), which are both super schemas of the actual MFS (1,1,-,-). Furthermore, no schemas obtained by this strategy have any relationship with the actual MFS (-,1,1,1), which means it was ignored.

It is noted that the motivating example in section 2 actually adopted this strategy. As a result, the schemas identified for Ex 2: (-,2,4,5), (-,3,4,5) are the super-schemas of the correct MFS(-,-,4,5).

4.3. Masking effects for FII approaches

Based on previous analysis, even though exhaustive testing is conducted to obtain T_{F_m}, we cannot determine the MFS set because of the masking effects. For traditional MFS identification approaches, i.e., FII approaches, masking effects can make problems worse.

This is because due to the time and computing rescouses limitation, FII approaches can only execute part of the whole test cases. Consequently, only part of T_{F_m} can be observed. Under this condition, the remaining of T_{F_m} need to be inferred. The inference process, which is based on the executed test cases and their outcomes, is key to the quality of the MFS identification result. Masking effects, however, not only makes $T_{mask(F_m)}$ be ignored as discussed before, but also significantly impact on the inference process.

Consider an FII example using the OFOT method [Nie and Leung 2011a]. Table IX gives an example to illustrate this approach. For SUT(2^4), assume the failing test case (1, 1, 1, 1) is being analysed, then OFOT approach can be illustrated as shown in Table IX. In this table, test case t failed, and OFOT mutated one parameter value of t at a time to generate additional test cases: $t_1; t_2; t_3; t_4$. The passings of t_1 and t_3 indicate that these two test cases break the MFS in the original test case t. So, (1,-,-,-) and (-,-,1,-) should be the failure-causing factors, and the other test cases (t_2, t_4) all failed, indicating that no other failure-inducing factors were broken (note that this conclusion is based on the assumption that t only contains one MFS). Therefore, the MFS in t is (1,-,1,-).

According to the MFS definition, all the test cases contain (1, -, 1, -) should fail. That is, test cases of $\mathcal{T}((1, -, 1, -))$ as listed in Table X should fail. Combining the observed failing test cases in Table IX, the failing test case inferred by OFOT in this case is (1, 0, 1, 0) as shown in Column 'inferred T_f' of Table X.

For the same OFOT example in Table IX, assume t_1 and t_2 (in bold) triggered other failures as shown in Table XI. To make the OFOT approach work properly in this case, without additional information, we need to apply the *regarded as one strategy* or

Table IX. OFOT example

test case under analysing					**Outcome**
t	1	1	1	1	Fail
additional test cases					
t_1	0	1	1	1	Pass
t_2	1	0	1	1	Fail
t_3	1	1	0	1	Pass
t_4	1	1	1	0	Fail
MFS					
	(1	-	1	-)	

Table X. Inferred test cases by OFOT

$\mathcal{T}((1,-,1,-))$	observed T_f	inferred T_f
(1, 0, 1, 0)		(1, 0, 1, 0)
(1, 0, 1, 1)	(1, 0, 1, 1)	
(1, 1, 1, 0)	(1, 1, 1, 0)	
(1, 1, 1, 1)	(1, 1, 1, 1)	

distinguishing failures strategy. In other word, t_1 and t_2 should either be determined as *Fail* or *Pass*.

Table XI. OFOT masking example

test case under analysing					**Outcome**
t	1	1	1	1	Fail
additional test cases					
$\mathbf{t_1}$	**0**	**1**	**1**	**1**	**Other Failure**
$\mathbf{t_2}$	**1**	**0**	**1**	**1**	**Other Failure**
t_3	1	1	0	1	Pass
t_4	1	1	1	0	Fail

The details using this two strategies are listed in Table XII, with the left part for *regarded as same failure* and the right part for *distinguishing failures*. For the first strategy, t_1 and t_2 are both determined as *fail*. As a result, only t_3 breaks the MFS. In this case, the MFS is considered to be (-, -, 1, -), which is the sub-schema of the original one. While for the second strategy, i.e., *distinguishing failures*, it determines t_1 and t_2 as *pass*. Under this condition, test cases t_1, t_2 and t_3 breaks the MFS; hence, the MFS is (1, 1, 1, -), which is the super-schema of the original MFS.

Such deviations for these two strategies are caused by the reduction of the observed test cases. Here, as t_1 and t_2 both triggered other failures, these two test cases cannot contribute to the MFS identification for the currently analysed failure. Besides this, the inference of OFOT are also affected. We list the inference test cases for these strategies in Table XIII. It can be found that both strategies cannot obtain the correct inference test cases as shown in Table X. Specifically, 4 test cases, i.e., (0, 0, 1, 0), (0, 0, 1, 1), (0, 1, 1, 0) and (0, 1, 1, 1), inferred by strategy *regarded as same failure* actually cannot trigger the current failure; while strategy *distinguishing failures* does not infer additional test cases (marked as '—').

As masking effect significantly impact on the FII approaches, alleviating this negative effect is desired to improve the quality of the identified MFS.

5. ILLUSTRATION OF THE APPROACH

The main reason that the FII approach fails to work properly is that it cannot determine $T_{mask(F_m)}$. In other words, if the test case triggers other unexpected failures which are different from the currently analysed F_m, it cannot figure out whether this test case will trigger F_m because of the masking effects. So to limit the impact of this

Table XII. OFOT with two strategies

regarded as same failure					
t	1	1	1	1	Fail
$\mathbf{t_1}$	**0**	**1**	**1**	**1**	**Fail**
$\mathbf{t_2}$	**1**	**0**	**1**	**1**	**Fail**
t_3	1	1	0	1	Pass
t_4	1	1	1	0	Fail
MFS					
(-	-	1	-)		

distinguishing failures					
t	1	1	1	1	Fail
$\mathbf{t_1}$	**0**	**1**	**1**	**1**	**Pass**
$\mathbf{t_2}$	**1**	**0**	**1**	**1**	**Pass**
t_3	1	1	0	1	Pass
t_4	1	1	1	0	Fail
MFS					
(1	1	1	-)		

Table XIII. Inferred test cases for two strategies

regarded as same failure

$\mathcal{T}((-,-,1,-))$	observed T_f	inferred T_f
(0, 0, 1, 0)		(0, 0, 1, 0)
(0, 0, 1, 1)		(0, 0, 1, 1)
(0, 1, 1, 0)		(0, 1, 1, 0)
(0, 1, 1, 1)		(0, 1, 1, 1)
(1, 0, 1, 0)		(1, 0, 1, 0)
(1, 0, 1, 1)		(1, 0, 1, 1)
(1, 1, 1, 0)	(1, 1, 1, 0)	
(1, 1, 1, 1)	(1, 1, 1, 1)	

distinguishing failures

$\mathcal{T}((1,1,1,-))$	observed T_f	inferred T_f
(1, 1, 1, 0)	(1, 1, 1, 0)	—
(1, 1, 1, 1)	(1, 1, 1, 1)	

effect on the FII approach, it is important to reduce the number of test cases that trigger other different failures, as this can reduce the probability that the expected failure may be masked by other different failures.

For the exhaustive testing model, i.e., $\mathcal{C}(T_{F_m})$, as all the test cases will be used to identify the MFS, there is no room left to improve its performance unless we fix other different failures and re-execute all the test cases. However, if only a subset of all test cases is used to identify the MFS (which is how the traditional FII approach works), it is important to make the right selection to limit the size of $T_{mask(F_m)}$ to be as small as possible.

5.1. Replacing test cases that trigger unexpected failures

The basic idea is to pick the test cases that trigger other failures and generate new test cases to replace them. The newly generated test case should either pass in the execution or trigger F_m.

Normally, when we replace the test case that triggers an unexpected failure with a new test case, we should keep some part of the original test case. We call this part the *fixed part*, and mutate the other part with different values from the original one. For example, assume that a test case (1,1,1,1) triggered an unexpected failure, 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 the same failure as currently analysed.

The *fixed part* can vary for different FII approaches. For example, for the OFOT [Nie and Leung 2011a] algorithm, all parameter values are the fixed part except for the one that needs to be validated, while for the FIC_BS [Zhang and Zhang 2011] approach, the fixed parts can be dynamically changed, depending on the outcome of the execution of last generated test case.

In this paper, we randomly select one test case from the candidate test case set, i.e., set of test cases that have the same fixed parts, and executed it to check whether it satisfies our requirement (either pass or trigger the same failure under analysis). This replacement process may need to be executed multiple times for one fixed part as it may not always be possible to coincidentally find a satisfied test case. The complete process of replacing a test case with a new one while keeping some fixed part is depicted in Algorithm 1.

ALGORITHM 1: Replacing test cases triggering unexpected failures

Input: failure F_m, all the candidate test cases R
Output: t_{new} the regenerate test case

```
1  while not R is empty do
2  |  t_new ← random_pick(R);
3  |  result ← execute(t_new);
4  |  R ← R\t_new;
5  |  if result == PASS or result == F_m then
6  |  |  return t_new;
7  |  else
8  |  |  continue;
9  |  end
10 end
11 return null
```

The inputs to this algorithm consist of the failure F_m under analysis, and the candidate test cases $R = \mathcal{T}(fixed) \backslash (T_{executed} \bigcup T_{invalid})$, where $\mathcal{T}(fixed)$ represents all the test cases that contain this fixed part, $T_{executed}$ represents those executed test cases and $T_{invalid}$ indicates that the test cases that does not satisfy the specified constraints. The output of this algorithm is a test case t_{new} which either triggers the expected F_m or passes.

The outer loop of this algorithm (lines 1 - 10) contains two parts:

The first part (lines 2 - 3) randomly generates and executes a new test case from the candidate test case set R. When a test case is generated, we remove it from the candidate test case set R to avoid redundancy (line 3).

The second part (lines 5 - 9) checks whether the newly generated test case is as expected. Specifically, if the test case passes or triggers the same failure – F_m, a satisfied test case is obtained (line 5) and returned (line 6). Otherwise, we will repeat the process, i.e., generate a new test case and check again (lines 7 - 8).

5.2. MFS identification with replacement strategy

With this replacement process, its easy to illustrate the complete MFS identification approach. Algorithm 2 shows the procedure of our approach. The inputs to this algorithm are the failure F_m that is currently focused on, an original failing test case t, and the specified constraints $Cons$. The output of this algorithm is the MFS for failure F_m in the test case t. $T_{executed}$ is the set of test cases that have already been executed, and initially has one element – test case t (line 1). $T_{executed}$ is used to avoid duplicate test cases. T_{judge} is the set of test cases that are used to be identify the MFS, which also has the original failing test case t (line 2). $T_{invalid}$ is the set of test cases which do not satisfy constraints $Cons$ (line 3). Note that here we just give a formal description of how to handle the constraints in CT. More details and implementations can be seen in works like [Cohen et al. 2007a; 2008; Yu et al. 2015].

Our approach loops until T_{judge} has enough test cases to determine which schemas in test case t are MFS (line 5 - 8). In each iteration, one test case $t_{current}$ is obtained from the traditional MFS identification approach line (lin 9), executed (line 10), and appended in $T_{executed}$ (line 11). If $t_{current}$ passes or fails with F_m (line 19), it will be added in the set of test cases T_{judge} (line 20). Otherwise, we will start the replacement procedure (line 12 - 18) to obtain a new test case $t_{replace}$ which either passes or fails with F_m. Specifically, we will first get the *fixed part* of $t_{current}$ which will not changed in the newly generated test case (line 13). Then we will obtain the set of candidate cases for selection (line 14). After using the replacement approach introduced in Algorithm

1 (line 16), we will added this newly generated test case $t_{replace}$ in T_{judge} (line 17). Additionally, the test cases generated and executed for this replacement (Those test cases which have already been used and eliminated from candidate test case set R) will be added in $T_{executed}$ (line 18).

ALGORITHM 2: MFS identification with Replacing test cases strategy

Input: failure F_m, original failing test case t, Constraints $Cons$
Output: MFS returning the determined MFS

```
1  T_executed ← {t} ;
2  T_judge ← {t} ;
3  T_invalid ← getInvalidTestCases(Cons) ;
4  while true do
5      if isEnd(T_judge) then
6          MFS ← identified(T_judge) ;
7          break;
8      end
9      t_current ← getTestCaseForExecution() ;
10     result ← execute(t_current);
11     T_executed.append(t_current);
12     if result ≠ PASS and result ≠ F_m then
13         fixed ← getCurrentFixedPart(t_current);
14         R ← T(fixed)\(T_executed ⋃ T_invalid) ;
15         O = R.copy() ;
16         t_replace ← Replacing(F_m, R);
17         T_judge.append(t_replace) ;
18         T_executed.appendAll(O\R) ;
19     else
20         T_judge.append(t_current);
21     end
22 end
23 return MFS
```

5.3. A case study using the replacement strategy

In this section, we will give a case study to illustrate our approach. This case study is based on the example shown in Section 4.3. In specific, assume the test case t_0 – (1, 1, 1, 1) fails after execution, and we need to identify the MFS in it. Additionally, we assume each parameter of the SUT has four values, i.e, $SUT(4^4)$, so that we can have more candidate values for replacement. Next, we will use our approach, i.e., traditional FII approach augmented with replacement strategy, to identify the MFS (Note that in this example, we use OFOT [Nie and Leung 2011a] as the FII approach, but our approach can be also applied on other MFS identification approaches).

The completed MFS identifying process listed in Table XIV works as follows. In this table, the underline part for each test case is the $fixed$ part according to OFOT [Nie and Leung 2011a]. We can learn that, the $fixed$ part of each test case generated by OFOT has three parameter values, only one parameter value is not *fixed*, i.e., can be replaced with other values.

The test case are generated and executed according to Algorithm 2. That is, if additionally generated test case does not trigger other failures (i.e., t'_1,t''_2,t_3,t'_4), the original FII process will continue until the MFS is identified. Otherwise, the replacement procedure starts when an unexpected failure is triggered. The replacement process will mutate the parameter values that are not in the $fixed$ part. For example, t'_1 changes

Table XIV. OFOT with replacement approach

test case under analysing					**Outcome**
t_0	1	1	1	1	Fail
additional test cases					
$\mathbf{t_1}$	**0**	**1**	**1**	**1**	**Other Failure**
t_1'	2	1	1	1	Pass
$\mathbf{t_2}$	**1**	**0**	**1**	**1**	**Other Failure**
$\mathbf{t_{2'}}$	**1**	**2**	**1**	**1**	**Other Failure**
t_2''	1	3	1	1	Fail
t_3	1	1	0	1	Pass
t_4	1	1	1	0	Fail
MFS					
(1	-	1	-)		

the first parameter value from 0 of t_1 to 2. Note that here the duplicated test cases will be eliminated. Hence, we can not change 0 to be 1 , because $(1, 1, 1, 1)$ has already been executed. Also noted that there may needs more than 1 trial to get the satisfied test case. For example, when find t_2 fails with other failure, we repeated the replacing procedure for two times, until t_2'' is obtained.

At last, all these test cases which either fails with expected failure or passes are appended in T_{judge}, i.e., $T_{judge} = \{t_1', t_2'', t_3, t_4\}$. Then we can learn that (1, -, 1, -) is the MFS according to OFOT (This is because only we changed the first and third parameter value, the test case passes, which means that these two parameter values break the MFS). This conclusion coincide with the correct result as said in Section 4.3.

Another notable point is that, in this example, test cases t_1, t_2, and t_3 do not contribute to the MFS identification. In a real-world scenario, however, it is appealing to iteratively start MFS identification for other failures based on these test cases.

6. DISCUSSION ABOUT THE INFLUENCE OF THE ASSUMPTIONS

In section 3.1, we introduced three assumptions that can simplify our formal modeling and proposed approach. In this section, we will discuss their influence on our propositions and approach, as well as some measures to alleviate their impact.

6.1. Deal with non-deterministic problem

The first assumption is that the outcomes of all the tests are deterministic. In practice, re-executing the same test case may result in different outcomes. For example, if the program using some random variables, different runs of the test case will assign different values to these random variables. As a result, the control flow of the program may be changed and hence the outcome will be different. We called this type of failure the non-deterministic failure [Yilmaz et al. 2006; Fouché et al. 2009]. Non-deterministic failure will complicate the MFS identification, and even worse, it may lead to an unreliable result of the MFS identification. Consider the the following example in Table XV, the only difference between the left and the right test set is the outcome of test case (0, 1, 1). This subtle difference leads to different results of the MFS identification. Hence, if there is one or more test cases which have non-deterministic executing results, the MFS identification is not reliable.

Inspired by the idea that using multiple same-way covering arrays to identify the MFS [Fouché et al. 2009], one potential solution to alleviate this non-deterministic failure is by adding redundancy, i.e., through re-executing the same test case to obtain the relatively stable outcome. However, this measure will increase the overall cost of the MFS identification, so the tradeoff between cost and the quality of MFS identification should be further studied.

Table XV. MFS of two similar test sets

T_{fail}	T_{pass}	MFS
(0, 0, 0)		(0, 0, -)
(0, 0, 1)		(0, -, 0)
(0, 1, 0)		
	(0, 1, 1)	
	(1, 0, 0)	
	(1, 0, 1)	
	(1, 1, 0)	
	(1, 1, 1)	

T_{fail}	T_{pass}	MFS
(0, 0, 0)		(0, -, -)
(0, 0, 1)		
(0, 1, 0)		
(0, 1, 1)		
	(1, 0, 0)	
	(1, 0, 1)	
	(1, 1, 0)	
	(1, 1, 1)	

6.2. Deal with failures distinguishing problem

The second assumption is that different errors in the software can be easily distinguished by information such as exception traces, state conditions, or the like. If we cannot directly distinguish them, our approach does not work. This is because we cannot determine which test case should be replaced and with what. In such case, one potential solution is to use the clustering techniques to classify the failures according to available information [Zheng et al. 2006; Jones et al. 2007; Podgurski et al. 2003]. If we cannot classify them because we do not have enough information (e.g., the black box testing) or it is too costly, we believe the only approach is to take the *regarded as same failure* strategy. With this strategy, we must be aware that the MFS identified are likely to be sub-schemas or irrelevant schemas of the actual MFS. Note that in the next section (empirical studies), all the failures can be distinguished from each other according to the exception traces.

7. EMPIRICAL STUDIES

To investigate the impact of masking effects on FII approaches in real software testing scenarios and to evaluate the performance of our approach in handling this effect, we conducted several empirical studies. Each of the studies focuses on addressing one particular issue, as follows:

Q1: Do masking effects exist in real software that contains multiple faults?

Q2: How well does our approach perform compared to traditional approaches?

Q3: Compared to the masking effects handling approach FDA-CIT [Yilmaz et al. 2014], does our approach have any advantages ?

7.1. The existence and characteristics of masking effects

In the first study, we surveyed seven kinds of open-source software systems to gain an insight into the existence of multiple faults and their effects. The software under study were Hsqldb, Jflex, Grep, Commons Cli, Joda Time, Commons Lang, and Jsoup. The first is a database management software written in pure Java, the second is a lexical analyser generator, and the last one is a command-line utility for searching plain-text data sets for lines matching a regular expression. The reason that we chose these systems is because they contain different versions and are all highly configurable so that the options and their interactions can affect their behaviour. Additionally, they all have a developer community so that we can easily obtain the real bugs reported in the bug tracker forum. Table XVI lists the program, the versions surveyed, number of lines of code, number of classes in the project, and the bug's ids [4] for each of the software.

[4]http://sourceforge.net/p/hsqldb/bugs
http://sourceforge.net/p/jflex/bugs
http://savannah.gnu.org/bugs/

Table XVI. Software under survey

software	info	versions	LOC	classes	bugs' id
Hsqldb	database management software	2.0rc8	139425	495	#981 & #1005
	written in pure Java	2.2.5	156066	508	#1173 & #1179
		2.2.9	162784	525	#1286 & #1280
Jflex	lexical analyser generator	1.4.1	10040	58	#87 & #80
		1.4.2	10745	61	#98 & #93
Grep	command-line utility for	2.12	27046	156	#7600 & #29537
	searching plain-text data	2.22	48101	297	#33080 & #28588
Cli	parsing command line	1.3.1	6433	48	#265 & #255
	options passed to programs	1.2	4630	44	#230 & #213
Joda	de facto standard date	2.8.2	85026	329	#297 & #296
	and time library for Java	2.9.1	85512	330	#361 & #347
		2.3	82158	317	#77 & #86
Lang	provides these extra methods for	3.4	66047	281	#1205 & #1215
	manipulation of core classes.	3.2	61596	247	#1087 & #1081
Jsoup	Java HTML Parser	1.8.3	10295	55	#688 & #689
		1.9.1	10489	56	#652 & #611

7.1.1. Study setup. We first looked through the bug tracker forum and focused on the bugs which are caused by the options interactions. For each of them, we derived its MFS by analysing the bug description report and the associated 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 interaction for this bug is (*preparestatement*, *placeHolder*, *Long string*). These three parameter values together form the condition that triggers the bug. The analysed result was referred to as the "prior MFS" later.

We further built the testing file for each version of the software listed in Table XVI. The testing file is constructed so that we can reproduce different failures by controlling the inputs to the test file. For each version, the source code of the testing file as well as other detailed information is available at http://gist.nju.edu.cn/doc/multi/.

Next, we built the input model which consists of the options related to the failure-inducing interactions and additional options that are commonly used. The detailed model information is shown in Table **??** for HSQLDB, JFLex and Grep, respectively. Each table is organised into three groups: (1) *common options*, which lists the options as well as their values under which every version of this software can be tested; (2) *specific options*, under which only the specific version can be tested; and (3) *configure space*, which depicts the input model for each version of the software, presented in the abbreviated form $\#values^{\#number\ of\ parameters} \times ...$, e.g., $2^9 \times 3^2 \times 4^1$ indicates the software has 9 parameters that can take 2 values, 2 parameters 3 values, and only one parameter 4 values.

We then generated the exhaustive test set consisting of all possible interactions of these options. For each of them, we executed the prepared testing file. We recorded the output of each test case [5] to observe whether there were test cases containing prior MFS that did not produce the corresponding bug. Later we refer to those test cases that contain the MFS but did not trigger the expected failure as the *masked* test cases.

[5]A test case is a test configuration (consists of the assignments to the options of the SUT) in our experiments. The testing file of each software will be executed with given test configurations.

Table XVII. Number of failures and their masking effects

software	versions	all tests	failure			masking	total
Hsqldb	2rc8	18432	#1(2304)	#2(1152)	#3(1152)	#1>#2#3(768)	768 (16.7%)
-	2.2.5	6912	#1(1728)	#2(1728)	-	#1>#2(576)	576 (16.7%)
-	2.2.9	6912	#1(2304)	#2(768)	#3(384)	#1>#2#3(960) #2>#3(768)	1728 (50%)
Jflex	1.4.1	36864	#1(12288)	#2(12288)	-	#1>#2(6144)	6144 (25%)
-	1.4.2	73728	#1(18432)	#2(18432)	-	#1>#2(6144)	6144 (16.7%)
Grep	2.12	384	#1(128)	#2(64)	#3(72)	#1>#2#3(80) #2>#3(16)	96 (36.4%)
-	2.22	576	#1(192)	#2(64)	#3(80)	#1>#2#3(80) #2>#3(16)	6144 (28.6%)
Cli	1.3.1	256	#1(32)	#2(64)	#3(40)	#1>#3(8) #2>#3(16)	24(17.6%)
-	1.2	256	#1(192)	#2(16)	-	#1>#2(48)	48 (23.1%)
Joda	2.8.2	1440	#1(288)	#2(288)	-	#1>#2(72)	72 (12.5%)
-	2.9.1	864	#1(216)	#2(72)	-	#1>#2(24))	24 (8.3%)
-	2.3	2304	#1(576)	#2(216)	-	#1>#2(72)	72 (9.1%)
Lang	3.4	432	#1(108)	#2(108)	-	#2>#1(36)	36 (16.7%)
-	3.2	576	#1(144)	#2(108)	-	#1>#2(36)	36 (14.3%)
Jsoup	1.8.3	576	#1(192)	#2(64)	#3(80)	#1>#2#3(80) #2>#3(16)	6144 (28.6%)
-	1.9.1	384	#1(96)	#2(96)	#3(96)	#1>#3(48) #2>#3(48)	96(33.3%)

7.1.2. Results and discussion. Table XVII lists the results of our survey. Column "all tests" gives the total number of test cases executed. Column "failure" indicates the number of test cases that failed during testing. Specifically, we give the specific number of failing test cases of each failure (labeled in the form #n(m), in which n indicates the nth failure, and m indicates the number of failing test cases with respect to this failure.). Column "masking" indicates the specific number of test cases (in the parentheses) that are masked by unexpected failure. In this column, we use the form (#m > #n#n'...) to indicate that failure #m masks failures (#n#n'...). The last column "total" shows the number of *masked* test cases in total (for all types of failures). The percentage in the parentheses in this column indicates the proportion of *masked* test cases and the failing test cases.

We observed that for each version of the software under analysis listed in Table XVII, test cases with masking effects do exist, i.e., test cases containing MFS did not trigger the corresponding bug. In fact, there are 768 out of 4608 test cases (about 16.7%) in hsqldb with 2rc8 version. This rate is about 16.7%, 50%, 25%, 16.7%, 36.4%, and 28.6% respectively, for the remaining software versions.

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

It is notable that in Yilmaz's [Yilmaz et al. 2014] paper, a similar study about the existence of the masking effects has been conducted. The main difference between that work and ours is that their work quantifies the impact of the masking effects as the number of τ-degree schemas that only appear in the test cases that triggered other failures. Here, the τ-degree schemas may not be MFS. Our work, however, quantifies the masking effects as the number of test cases that are masked by different failures. These test cases should contain some MFS, i.e., they should have triggered the expected failure if they did not trigger any other different failure. The reason that we quantify the masking effects in such way is because our work seeks to overcome the masking effects in the MFS identifying process. As these test cases which contain the MFS but fail to produce the corresponding failure will significantly affect the MFS identifying results, their number can better reflect the impact of the masking effects on the FII approach.

7.2. Comparing our approach with traditional approaches

The second study aims to compare the performance of our approach with traditional approaches in identifying MFS under the impact of masking effects. To conduct this study, we need to apply our approach and traditional algorithms to identify MFS in a variety of software and evaluate their results. The seven versions of software in Table XVI used as test objects are far from the requirement for a general evaluation. However, to construct real testing objects for evaluations is time-consuming. This is because we must carefully study the detail of software systems as well as their bug tracker reports. To compromise, we synthesized 5 more testing objects. These synthesized objects are five small programs which work as follows: 1) it first judges whether a given test case contain some prior MFS, 2) and then outputs a failure with respect to the MFS it contains (if there exists multiple MFS in one test case, output the failure of the MFS which will not be masked by others) (details of them are also available at http://gist.nju.edu.cn/doc/multi/). These synthesized objects are created such that their testing models are similar to those real systems, i.e., to make the number of parameters, the number of failures, and the possible masking effects similar to that of those real ones listed in Table XVII. Specifically, we limited the number of parameters of the SUT ranged from 7 to 15, the number of different failures in the SUT ranged from 2 to 5, and the number of MFS of a failure ranged from 1 to 2, in which the degree of the MFS ranged from 1 to 3.

Table XVIII lists the testing model for both the real and synthetic testing objects. In this table, column 'Object' indicates the SUT under test. For the real SUT listed in Table XVI, we label the seven software as $H2cr8$, $H2.2.5$, $H2.2.9$, $J1.4.1$, $J1.4.2$, $G2.12$ and $G2.22$ respectively. While for the synthesized ones, we label them in the form of '*syn+ id*'. Column 'Model' presents the input space for each testing object. Column 'Failures' shows the different failures in the software and their masking relationships. In this column, '$>$' means the left failure can mask the right failure, i.e., if a failure of the left side is triggered, then the failure of the right side will not be triggered. Furthermore, '$>$' is transitive so that the left failure can mask all the failures in the right. For example, for the $H2cr8$ object, we can find three failures : e_1, e_2, and e_3. The notation of $e_1 > e_2 > e_3$ indicates that failure e_2 will mask e_3; and e_1 will mask both e_2 and e_3. Here for the simplicity of the experiment, we did not build more complex testing scenarios such as masking effects in the form $e_1 > e_2,\ e_2 > e_3,\ e_3 > e_1$ or even $e_1 > e_2,\ e_2 > e_1$. The last column shows the MFS of each failure. The MFS is presented in an abbreviated form $\{\#index_{\#value}\}_{failure}$, e.g., for the object $H2cr8$, $(5_1, 6_0, 7_0)_{e_1}$ actually means (-, -, -, -, -, 1, 0, 0, -, -, -, -) is the MFS of failure e_1.

7.2.1. Study setup. After preparing the objects under testing, we then applied our approach (FIC_BS with replacement strategy) to identify the MFS. Specifically, for each SUT we selected each test case that failed during testing and fed it into our FII approach as the input. Then, after the identifying process was completed, we recorded the identified MFS and the extra test cases needed. For the traditional FIC_BS approach, we designed the same experiment. But as the objects being tested have multiple faults for which the traditional FIC_BS can not be applied directly, we adopted two traditional strategies on the FIC_BS algorithm, i.e., *regarded as same failure* and *distinguishing failures* as described in Section 4.3. The purpose of recording the generated additional test cases is to 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 approach suffers from masking effects. There are five metrics used in this study, listed as follows:

(1) *Accurate number* : the number of identified MFS which are actual prior MFS.

Table XVIII. The testing models used in the case study

Object	Model	Failures	MFS of each failure
Hsql1	$2^9 \times 3^2 \times 4^1$	$e_1 > e_2 > e_3$	$(5_1, 6_0, 7_0)_{e_1}, (5_1, 8_2, 9_2)_{e_2}, (5_1, 8_2, 9_1)_{e_2}, (5_1, 8_3, 9_2)_{e_3}, (5_1, 8_3, 9_1)_{e_3}$
Hsql2	$2^8 \times 3^3$	$e_1 > e_2$	$(6_1, 7_0)_{e_1}, (5_2)_{e_2}$
Hsql3	$2^8 \times 3^3$	$e_1 > e_2 > e_3$	$(6_0)_{e_1}, (0_1, 5_1, 7_0)_{e_2}, (0_0, 5_1, 7_0)_{e_2}, (5_1, 7_0)_{e_3}$
Jflex1	$2^{10} \times 3^2 \times 4^1$	$e_1 > e_2$	$(0_0)_{e_1}, (1_0)_{e_2}$
Jflex2	$2^{11} \times 3^2 \times 4^1$	$e_1 > e_2$	$(1_0, 2_1)_{e_1}, (0_1)_{e_2}$
Grep1	$2^5 \times 3^1 \times 4^1$	$e_1 > e_2 > e_3$	$(0_0)_{e_1}, (1_1, 2_1)_{e_2}, (3_0, 4_1)_{e_3}, (3_1, 4_1)_{e_3}, (3_2, 4_1)_{e_3}$
Grep2	$2^4 \times 3^2 \times 4^1$	$e_1 > e_2 > e_3$	$(0_0)_{e_1}, (1_0, 2_3)_{e_2}, (1_1, 2_3)_{e_2}, (3_0, 4_0)_{e_3}$
Cli1	2^8	$e_1 > e_2 > e_3$	$(5_0, 6_0, 7_1)_{e_1}, (6_1, 7_1)_{e_2}, (3_1, 4_0)_{e_3}$
Cli2	2^8	$e_1 > e_2$	$(7_0)_{e_1}, (6_1)_{e_1}, (3_1, 4_0)_{e_2}$
Joda1	$2^5 \times 3^2 \times 5^1$	$e_1 > e_2$	$(6_2)_{e_1}, (4_1, 5_1)_{e_2}$
Joda2	$2^3 \times 3^3 \times 4^1$	$e_1 > e_2$	$(6_3)_{e_1}, (4_2, 5_2)_{e_2}$
Joda3	$2^6 \times 3^2 \times 4^1$	$e_1 > e_2$	$(6_0, 7_1)_{e_1}, (4_0, 5_1)_{e_2}$
Lang1	$2^4 \times 3^3$	$e_1 > e_2$	$(3_0)_{e_1}, (5_1, 6_1)_{e_2}$
Lang2	$2^6 \times 3^2$	$e_1 > e_2$	$(6_1, 7_1)_{e_1}, (3_1, 4_1)_{e_2}$
Jsoup1	$2^8 \times 3^1$	$e_1 > e_2$	$(5_1, 6_1)_{e_1}, (5_2, 6_1)_{e_1}, (5_0, 7_1)_{e_1}, (5_1, 7_1)_{e_1}, (5_0, 8_1)_{e_1}, (5_1, 8_1)_{e_1}, (3_1, 4_0)_{e_2}$
Jsoup2	$2^6 \times 6^1$	$e_1 > e_2 > e_3$	$(5_0, 6_1)_{e_1}, (5_1, 6_1)_{e_2}, (4_1)_{e_3}$

(2) *Super number*: the number of identified MFS that are the super schemas of some prior MFS.
(3) *Sub number* : the number of identified MFS that are the sub schemas of some prior MFS.
(4) *Ignored number* : the number of schemas that are in the prior MFS, but irrelevant to the identified MFS.
(5) *Irrelevant number* : the number of schemas in the identified MFS that are irrelevant to the prior MFS.

Among these five metrics, the *accurate number* directly indicates the effectiveness of the FII approaches, since the target for every FII approach is to identify as many actual MFS as possible. Metrics *ignored number* and *irrelevant number* indicate the extent of deviation for the FII approaches. Specifically, the former indicates how much information about the MFS will miss, while the latter indicates how serious the distraction would be due to the "useless" schemas identified by the FII approach. *Super number* and *sub number* are the metrics in between, i.e., to identify some schemas that is *super* or *sub* schemas of the actual MFS is better than identifying *irrelevant* ones or ignoring some MFS, but it is worse than identifying the schema that is identical to the actual MFS. This is intuitive, as given the *super* / *sub* schemas, we just need to *remove* / *add* some elements of the original schemas to get the actual MFS. While for the *irrelevant* or *ignore* schemas, however, more efforts will be needed (e.g., both *adding* and *removing* operations will be needed to revise the irrelevant schemas to the actual MFS).

Besides these specific metrics, we also define a composite metric to measure the overall performance of each approach. The composite metric *aggregate* is defined as follows:

$$Aggregate = \frac{accurate + related(super) + related(sub)}{accurate + super + sub + irrelevant + ignored}$$

In this formula, *accurate, super, sub, irrelevant,* and *ignored* represent the value of specific metric. To refine the evaluation of different *super* / *sub* schemas, we design a

related function which gives the similarity between the schemas (either super or sub) and the real MFS, so that we can quantify the specific effort for changing a *super* / *sub* schema to the real MFS. The similarity between two schemas c_1 and c_2 is computed as:

$$Similarity(c_1, c_2) = \frac{number\ of\ same\ elements\ in\ c_1\ and\ c_2}{\max(Degree(c_1), Degree(c_2))}$$

For example, the similarity of (- 1 2 - 3) and (- 2 2 - 3) is $\frac{2}{3}$. This is because (- 1 2 - 3) and (- 2 2 - 3) have the same third and last elements, and both of them are 3-degree.

The *related* function gives the summation of similarity of all the super or sub schemas with their corresponding MFS.

7.2.2. Results and discussion. Figure 2 depicts the results of the second case study. There are seven sub-figures in this figure, i.e., Figure 2(a) to Figure 2(g). They indicate the results of the number of accurate MFS each approach identified, the number of identified schemas which are the sub-schema / super-schema of some prior MFS, the number of ignored prior MFS, the number of identified schemas which are irrelevant to all the prior MFS, the aggregate value, and the extra test cases each algorithm needed, respectively.

In each sub-figure, there are four polygonal lines, each of which shows the results for one of the four strategies: *regarded as same failure*, *distinguishing failures*, *replacement strategy based on ILP searching*, *replacement strategy based on random searching* (The last one will be discussed in the next case study). Specifically, each point in the polygonal line indicates the specific result of a particular strategy for the corresponding testing object. For example in Figure 2(a), the point marked with '■' at (1,0.25) indicates that the approach using *regarded as same failure* strategy identified 0.25 accurate MFS on average for each failing test case of the testing object–HSQLDB 2cr8. The raw data for this experiment can be found in Table **??** of the Appendix. Note that all these data are average values, i.e., the average performance of these approaches when identifying the MFS for each failing test case.

Accurate number: Figure 2(a) shows the average number of accurate schemas that each approach achieved. It appears that *ILP* performed the best among the three approaches. In fact, for most testing objects (testing objects 1, 4, 5, 6, 7, 9, 10, 11 and 12), *ILP* either obtained the most number of accurate MFS, or tie with the most number of accurate MFS. The second best approach is *distinguishing failures*, which obtained better results than *regarded as same failure* for testing objects 1, 6, 8, 9, 10, 11, and 12.

Sub number & super number: Figure 2(b) and 2(c) depict the results for *sub number* and *super number*, respectively. These two figures firstly showed a clear trend for strategies *regarded as same failure* and *distinguishing failures*, i.e., the former identified more sub schemas of actual MFS than the latter, while the latter identified more super schemas of actual MFS than the former. This is consistent with our formal analysis in Section 4.1 and Section 4.2.

The performance of our strategy *ILP* for these two metrics are in between. Specifically, ILP identify more sub schemas than strategy *distinguishing failures* but fewer than *regarded as same failure*; and ILP identify more super schemas than *regarded as same failure*, but fewer than *distinguishing failures*.

Ignore number & irrelevant number: The results of the two negative performance metrics are given in Figure 2(d) and 2(e), respectively. One observation is that, comparing with strategy *regarded as same failure*, *distinguishing failures* obtained fewer irrelevant schemas, but ignored more actual MFS. This is also consistent with

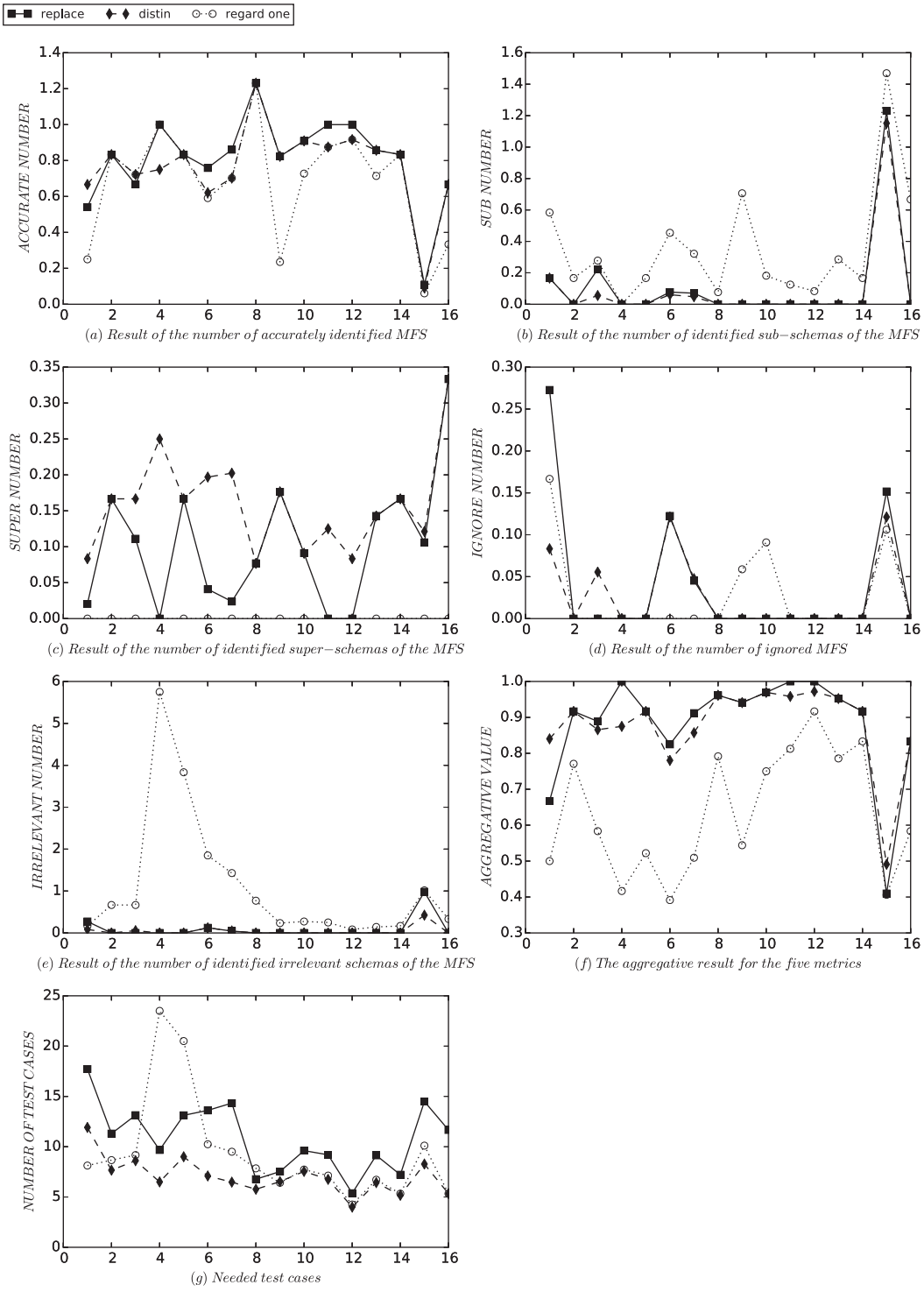


Fig. 2. Result of the evaluation for the second case study

the formal analysis in Section 4. Note that for metric *irrelevant number*, strategy *regarded as same failure* obtained significantly much more irrelevant schemas than the remaining strategies, which make it hard to distinguish each other if we post them together on one figure. Hence, we draw this figure with only three polygonal lines (for *distinguishing failures*, *ILP* , and *random* respectively). Besides this, we offer an additional smaller figure in Figure 2(e), which includes the strategy *regarded as same failure*, to depicts the comparison of all these approaches.

The second observation is that *ILP* did a good job at reducing the scores for these two negative metrics. Specifically, for *ignored number*, our approach performed better than strategy *distinguishing failures* for testing objects (1, 3, 7, 10, 11, 12) in Figure 2(d), but is not as good as strategy *regarded as same failure*. In fact, strategy *regarded as same failure* has a significant advantage at reducing the number of ignored MFS as it tends to associate the failures with all the failing test cases. However, when we consider the *irrelevant number*, our approach is the best among all three strategies (better than *distinguishing failures* at testing objects 1, 3, 6, 7, 11, 12 in Figure 2(e), and better than strategy *regarded as same failure* for almost all the testing objects) . We believe this improvement is caused by our test cases replacing strategy, as it can increase the test cases that are useful for identifying the MFS and decrease those useless test cases.

Aggregative for the five metrics: The composite results are given in Figure 2(f). This metric gives an overall evaluation of the quality of the identified schemas. From this figure, we can find that *ILP* performed the best, next the *distinguishing failures*, the last is the *regarded as same failure* (See the testing objects 1, 3, 4, 6, 7, 10, 11, and 12 in Figure 2(f)).

It is as expected that *ILP* performed better than *distinguishing failures* as it is actually the refinement version of latter. In fact, *ILP* also make the failures distinguished from each other. The main difference between *ILP* and *distinguishing failures* strategy is that the former has to replace the test cases that triggered any failure other than the currently analysed one while the latter will not change the generated test cases.

It is a bit of surprise to find, however, that strategy *distinguishing failures* performed better than *regarded as same failure* at almost all the testing objects. This result cannot be derived from the formal analysis. We believe the reason is that the masking effects are *monotonic* in the testing objects we constructed for evaluation. That is, our study includes only the case that bug A always mask bug B, and does not consider cases that bug A can mask bug B and bug B can also mask bug A. This condition is favorable for the *distinguishing failures* strategy. For example, assume bug A masks bug B; then when we identify the MFS of bug A, the *distinguishing failures* is the correct strategy, as if there is a test case trigger the bug B, then it must not trigger the bug A (otherwise, bug B will not be triggered). Hence, the probability that *distinguishing failures strategy* makes the correct operation is at least 50% .

Test cases: The number of test cases generated for identifying the MFS indicates the cost of FII approach. The result is shown in Figure 2(g). We can find that strategy *ILP* generated more test cases than the other strategies. Specifically, the gap between *ILP* and the other two strategies ranged from about 2 to 5. This is acceptable when comparing to all the test cases that each approach needed. The increase in test cases for our approach is necessary, as additional test cases must be generated when some test cases cannot help to identify the MFS of the currently analysed failure. As for strategies *distinguishing failures* and *regarded as same failure*, there is no significant difference between them.

Above all, we draw three conclusions, which help to answer **Q2**:

1) *Distinguishing failures* strategy obtained more *super number* and *ignored number* than *regarded as same failure* strategy, while the latter identified more *sub number*

and *irrelevant number* than the former. This result is consistent with the previous formal analysis in Section 4.

2) Considering the quality of the MFS each approach identified, we can find that our *ILP* approach achieves the best performance, followed by the strategy *distinguishing failures*.

3) Although our approach needs more test cases than the other two strategies, it is acceptable.

7.3. Comparison with Feedback driven combinatorial testing

The *FDA-CIT* [Yilmaz et al. 2014] approach handles masking effects by generating covering array that can cover all the τ-degree schemas without being masked by the MFS. There is an integrated FII approach in the FDA-CIT, which has two versions, i.e., *ternary-class* and *multiple-class*. In this paper, we use the multiple-class version for comparison, as it performs better than the former [Yilmaz et al. 2014].

The FDA-CIT process starts with generating a covering array (In [Yilmaz et al. 2014], this is a test case-aware covering array [Yilmaz 2013]). After executing the test cases in this covering array, it records the outcome of each test case and then applies the classification tree method on the test cases to characterize the MFS of each failure. It then labels these MFS as the schemas that can trigger masking effects. Later, if the interaction coverage is not satisfied (here the interaction coverage criteria is different from the traditional covering array [Yilmaz et al. 2014]), it will re-generate a covering array that aims to cover these schemas that were masked by the MFS and then repeat the previous steps.

The main target of FDA-CIT is to guarantee that the generated test cases should cover all the τ-degree schemas. To achieve this goal, FDA-CIT needs to repeatedly identify the schemas that can trigger the masking effects. So to make the two approaches (FDA-CIT and ILP) comparable, we need to collect all the MFS that FDA-CIT characterized in each iteration and then compare them with the MFS identified by our approach.

7.3.1. Study setup. As FDA-CIT used a post-analysis (classification tree) technique on covering arrays, we first generated 2 to 4 ways covering arrays. The covering array generating method is based on augmented simulated annealing [Cohen et al. 2003], as it can be easily extended with constraint dealing and seed injection [Cohen et al. 2007b], which is needed by the FDA-CIT process. As different test cases will influence the results of the characterization process, we generated 30 different 2 to 4 way initial covering arrays. Then before we feed them to the FDA-CIT approach, we extend the corresponding τ-way covering array to τ+1-way covering array. This is because FDA-CIT needs higher-way covering arrays to identify MFS [Yilmaz et al. 2014]. The classification tree method which is integrated into FDA-CIT is implemented by Weka J48 [Hall et al. 2009], of which the configuration options are set as follows: the n-fold of cross validation is set to 10, the accuracy cutoff is set to be 1 and the default confidence factor is 0.25 [Yilmaz et al. 2006]. After running FDA-CIT, we recorded the MFS identified, and by comparing them with prior actual MFS, we can evaluate the quality of the identified schemas according to the metrics mentioned in the previous case study.

Besides the FDA-CIT, we also applied our ILP-based approach to the initial τ-way covering array. Specifically, for each failing test case (except those test cases which contain existed identified MFS) in the covering array, we separately applied our approach to identify the MFS of that case. In fact, we can reduce the number of extra test cases if we utilize the other test cases in the covering array [Li et al. 2012]), but we did not utilize the information to simplify the experiment. Similarly, we then recorded the MFS identified by our approach, and evaluated them according to the corresponding metrics. In addition, we recorded the overall test cases (including the initially gener-

ated covering array) that this approach needed and compared the magnitude of these test cases with that of FDA-CIT.

7.3.2. Result and discussion. The result is shown in Figure 3. There are three sub-figures of Figure 3, which describe the results of the experiments based on 2-way, 3-way and 4-way covering arrays, respectively. In each sub-figure, there are 7 columns, showing the outcomes for the previous mentioned 6 metrics and one more metric (Column *Testcase*), which indicates the overall test cases that each approach needed. Each column has two bars, which indicate the results for approach FDA-CIT, and ILP respectively.

Note that in Figure 3, the results for each metric is the average evaluation for all the results of the experiments on the 12 testing objects in Table XVIII. The raw results (as well as the p-values) for each testing object are listed in Table **??** in the appendix. The raw data is organised the same way as Table **??**.

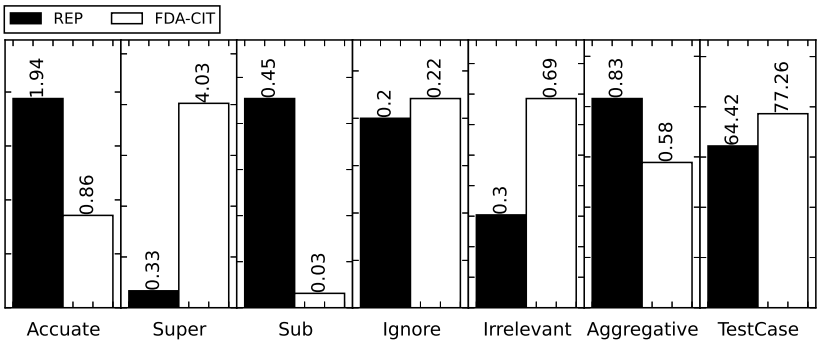
From Figure 3, we have the following observations:

First, for most metrics in our study, the performance of each approach is relatively stable against the change of degree τ. The only exceptions are metrics *ignore number* and *irrelevant number*. With increasing τ, the value of ignore number and irrelevant number of approach *FDA-CIT* decrease rapidly, while the performance of *ILP* is relatively steady when compared to *FDA-CIT*. Apart from these two metrics, the relationships between the two approaches for the other metrics are stable. Take for example the metric *accurate number*. No matter what τ is (2, 3 or 4), *ILP* always obtained more accurate schemas than *FDA-CIT*. This observation indicates that the difference between the performance of these approaches is not dependent on the characteristics of the covering array, but instead on the approaches themselves.

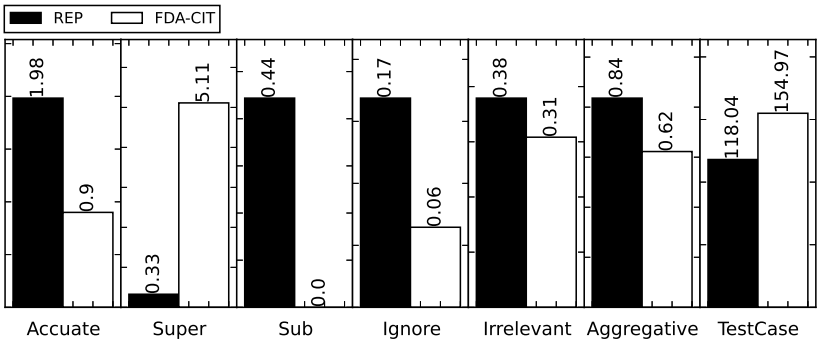
From the perspective of p-values, we can also get the same observation. For all the metrics except *ignore number* and *irrelevant number*, the p-values are relatively small. Note that there also exist some test objects that the p-values are *NaN*. This is because that the MFS of these testing objects are relatively simple (between 2 to 3 MFS). As a result, both approaches obtained the same schemas for the 30 tries. Besides the *NaN* values, other values of these metrics indicates that the difference between these two approaches are statistically significant. As for metrics *ignore number* and *irrelevant number*, the p-values are relatively large. For example, for the 2-way experiment, some p-values of metric *ignore* are around 0.3 (Column '*syn1*') and 0.1 (Column '*syn3*'), which show that there are no significant different between the two approaches for this metric.

Second, comparing to *ILP*, the schemas identified by *FDA-CIT* tend to be super schemas rather than sub schemas of the original MFS. We believe this result is due to the use of the classification tree analysis. Note that the way that CTA constructing decision trees usually work top-down, by choosing a parameter value at each step that best splits the set of test cases [Rokach and Maimon 2005]. After that, one path (conjunction of nodes from the root to a leaf in the tree) in this tree is deemed as an MFS. Consequently, the MFS identified by FDA-CIT tends to share some common nodes (here, a node represents a parameter value), e.g. the root node, which result in the identified schemas containing some unnecessary parameter value. This induces the so-called 'over fitting' problem. As a result, the schemas identified by *FDA-CIT* tend to be the super schemas of the actual MFS.

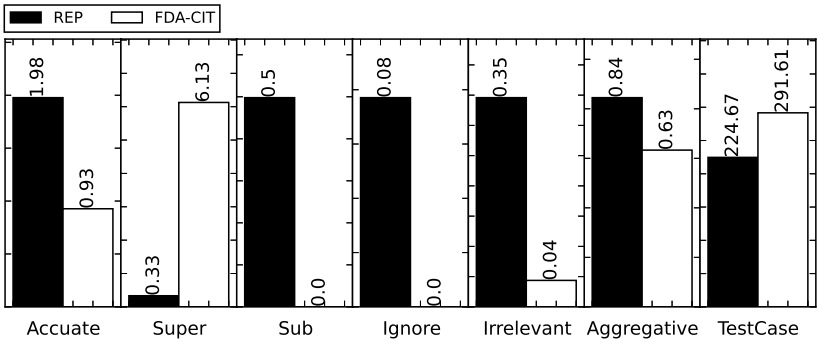
Third, in terms of the quality of the MFS identified, we can clearly find that our approach performed better than FDA-CIT. This is manifested in that our approach obtained more accurate schemas (the differences are statistically significant for all 3 degrees). We believe this gap is mainly caused by the FII approach. However, this result does not mean that FIC_BS is better than the classification tree method under all conditions. In fact, the classification tree method has its own advantage, i.e., it is



(a) Result for the 2-way covering array



(b) Result for the 3-way covering array



(c) Result of the 4-way covering array

Fig. 3. Three approaches augmented with the replacing strategy

flexible and does not need to generate additional test cases. As a result, FDA-CIT can be applied to some complex testing scenarios, e.g., occasional failures, and incompatibilities between test cases and configurations [Yilmaz et al. 2014].

At last, when considering the cost of these two approaches, our approach needs fewer test cases. However, it should be noted that FDA-CIT is conducted on τ+1 way covering arrays. Hence, FDA-CIT can get better performance when using τ-way covering array.

Above all, we can conclude three points in this experiment, which provide answer to **Q3**:

1)The degree τ of the covering array does not affect the overall performance of approaches, but for FDA-CIT, it may get better performance at reducing the *ignore number* and *irrelevant number* when using higher-τ-way covering array.

2)When taking account of the quality of MFS identification, *ILP* approach preforms better.

3)Without additional measures, FDA-CIT is a better choice when handling some complex testing scenarios.

It is noted that FDA-CIT's primary concern is to avoid masking effects and to give every τ-degree a fair chance to be tested, not to perform fault characterization. FDA-CIT is also an established technique which can work with non-deterministic failures and in the presence of inter-option constraints. So even though our approach *ILP* has shown a better performance at MFS identification, it is appealing and effective to use FDA-CIT to guide the test generation when taking into account the masking effects. In fact, considering FDA-CIT can work with other fault characterization approaches, it may also be appealing to combine these two approaches for MFS identification.

7.4. Threats to validity

7.4.1. internal threats. There are two threats to internal validity. First, the characteristics of the actual MFS in the SUT can affect the FII results. This is because the magnitude and location of the MFS can make the FII approaches generate different test cases. As a result, it can lead to different observed failing test cases and inferred failing test cases. In the worst case, the FII approach happens to identify the exact actual MFS, then our test case replacing strategy is of no use. In this paper, we used 12 testing objects, in which 5 are real software systems with real faults and 10 synthetic ones with injected faults. To reduce the influence caused by different characteristics of the MFS, we need to build more testing objects and inject additional types of failures for a more comprehensive study of our approach.

The second threat is that we just applied our test case replacing strategy on one FII approach – FIC_BS [Zhang and Zhang 2011]. Although we believe the test case replacing strategy can also improve the quality of the identified MFS for other FII approaches when the testing object is suffering from masking effects, the extent to which their results can be refined may vary for different FII approaches. For example, for FIC_BS [Zhang and Zhang 2011] used in this paper, there are about $(v-1)$ to $(v-1)^{k-1}$ (k is the number of parameters in a test case, v is the number of values each parameter can take) candidate test cases that can be replaced when one test case triggered other failures, while for OFOT [Nie and Leung 2011a], there are $(v-1)$ candidates. As a result, FIC_BS can have a higher chance than OFOT to find a satisfied test case. To learn the difference between the improvement of various FII approaches when applying our test case replacing strategy, we need to try more FII approaches in the future.

7.4.2. external threats. One threat to external validity comes from the real software we used. In this paper we have only surveyed three types of open-source software with seven different versions, of which the program scale is medium-sized. This may impact the generality of our results.

The second threat comes from the possible masking relationships between multiple faults in the real software. In this paper, we just focus on the condition that the masking effects are transitive, i.e., if fault A masks B, and fault B masks C, then fault A must mask fault C. In practice, the relationships between multiple faults may be more complicated. One possible scenario is that two faults are in a loop, for which they can even mask each other in a particular condition. Such a case will make our formal analysis invalid and will significantly complicate the relationships between schemas and

their corresponding test cases. A new formal model should be proposed to handle that type of masking effects.

The third threat is that all the failures in the experiments are option-related. In practice, some failures may not be related to the parameters modeled in the SUT. For example, consider the Internet Explorer option-compliant testing problem [Nie et al. 2013]. Initially we may not properly model the options we tested, as a result, it can happen that the explorer will always crash no matter we change which option in the initial model. This will cause the MFS identification invalid, as all the test cases fail during testing. To solve this problem, one potential solution is to re-model the options we should test, as the error may be related to other options in the SUT (For example, those options which are set to default value). Or alternately, we should try other testing techniques to assist original CT. For example, if the error is related to the internal code instead of those configuration options, we may try program slicing technique [Weiser 1981] or spectrum-based approaches [Naish et al. 2011]. In such case, the parameters that we model for the SUT should not only be limited to configuration options or simple inputs, but also those variables, predicates, or other logical structures and data in the software under test.

8. RELATED WORKS

Shi and Nie [Shi et al. 2005] presented an approach for failure revealing and failure diagnosis in CT , which first tests the SUT with a covering array, then reduces the value schemas contained in the failing test case by eliminating those appearing in the passing 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 failing test case, and the schema set is then further reduced, until no more faults are found or the fault is located. Based on this work, Wang [Wang et al. 2010] proposed an AIFL approach which extended the SOFOT process by adaptively mutating factors in the original failing test cases in each iteration to characterize failure-inducing interactions.

Nie et al. [Nie and Leung 2011a] introduced the notion of Minimal Failure-causing Schema(MFS) and proposed the OFOT approach which is an extension of SOFOT that can isolate the MFS in the SUT. This approach mutates one value of that parameter at a time, 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 can also detect the newly imported faulty interactions.

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

Colbourn and McClary [Colbourn and McClary 2008] proposed a non-adaptive method. Their approach extends a covering array to the locating array to detect and locate interaction failures. Martínez [Martínez et al. 2008; 2009] proposed two adaptive algorithms. The first one requires safe value as the assumption and the second one removes this assumption when the number of values of each parameter is equal to 2. Their algorithms focus on identifying faulty tuples that have no more than 2 parameters.

Ghandehari et al. [Ghandehari et al. 2012] defined the suspiciousness of tuple and suspiciousness of the environment of a tuple. Based on this, they ranked the possible

tuples and generated the test configurations. They [Ghandehari et al. 2013] further utilized the test cases generated from the inducing interaction to locate the fault.

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

Our previous work [Niu et al. 2013] proposed an approach that utilizes the tuple relationship tree to isolate the failure-inducing interactions in a failing test case. One novelty of this approach is that it can identify the overlapped faulty interaction. This work also alleviates the problem of introducing new failure-inducing interactions in additional test cases.

In addition to the studies that aim at identifying the failure-inducing interactions in test cases, there are others that focus on working around the masking effects.

Constraints handling becomes more and more popular in CT these years. A constraint is an invalid interaction that should not appear in the test case. It can be deemed as the masking effect which are known in prior [Yilmaz et al. 2014]. Cohen [Cohen et al. 2007a; 2007b; 2008] studied the impact of the constraints that render some generated test cases invalid in CT. They also proposed an approach that integrates the incremental SAT solver with the covering arrays generating algorithm to avoid those invalid interactions. Further study was conducted [Petke et al. 2013] to show that with consideration of constraints, higher-strength covering arrays with early failure detection are practical.

Besides, there are additional works that aim to study the impact of constraints for CT [Garvin et al. 2011; Bryce and Colbourn 2006; Calvagna and Gargantini 2008; Grindal et al. 2006; Yilmaz 2013]. Among them, [Bryce and Colbourn 2006] distinguished the constraints into two types: *hard* and *soft*, which the former cannot be included in the test case, while the latter can be permitted, but not desirable. [Grindal et al. 2006] comprehensively compared the performance of four strategies at handling the constraints in the covering array. [Calvagna and Gargantini 2008] proposed an heuristic strategy to handle the constraints. It can support an ad-hoc inclusion or exclusion of interactions such that the user can customize output of the covering array. [Garvin et al. 2011] refined the simulated annealing algorithm to efficiently construct the covering array while considering the constraints. [Yilmaz 2013] introduced the test case-specific constraints; differing from the system-wide constraints, these constraints can only be triggered in some specific test cases.

Chen et al. [Chen et al. 2010] 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. 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 [Dumlu et al. 2011]. proposed a feedback-driven approach to work around the masking effects Specifically, they first used classification tree to classify the possible failure-inducing interactions and eliminate them. Then they generate new test cases to detect possible masked interaction in the next iteration. They [Yilmaz et al. 2014] further extended their work by proposing a multiple-class CTA approach to distinguish failures in the SUT. In addition, they empirically studied the impact of masking effects on both ternary-class and multiple-class CTA approaches.

All works above can be categorized into 3 groups according to their relationships with our work. First, we discuss the works that aim to identifying the MFS in the SUT. Our work also focuses on identifying the MFS, but instead of single fault, our work considers the impact of multiple faults on the FII approaches, and based on this,

a test case replacement strategy is proposed that can assist these FII approaches in reducing the negative effects. Second, the works that aim to deal with the constraints. As discussed before the constraints can be deemed as a special masking effect. Our work differs from them in that the masking effects handled in this paper are those that can be dynamically triggered; that is, we did not know them in prior. Another difference between our work with these constraints handling works is that their target is to avoid the constraints when generating covering array. However, our work aims to remove the masking effects of the FII approaches. Last, the work[Yilmaz et al. 2014] that is most similar to our work, which also considered the masking effects that are dynamically appeared in test cases. But different from our work, it mainly focused on reducing the masking effects in the covering array, so that the covering array can support a comprehensive validation of all the τ-degree schemas. The approach used to reduce this negative effect is to use the FII approach to identify the schemas that can trigger this effect in each iteration. Our approach, however, addresses the masking effects that happened in these FII approaches themselves, and our approach alleviates the masking effects by augmenting the FII approaches with a test case replacement strategy.

9. CONCLUSIONS

Masking effects of multiple faults in the SUT can bias the results of traditional failure-inducing interactions identification approaches. In this paper, we formally analysed the impact of masking effects on FII approaches and showed that the two traditional strategies, i.e., *regarded as same failure* and *distinguishing failures*, are both inefficient in handling such impact. We further presented a test case replacement strategy for FII approaches to alleviate such impact.

In our empirical studies, we extended FIC_BS [Zhang and Zhang 2011] with our strategy. A comparison between our approach and traditional approaches was performed on several open-source software. The results indicate that our strategy assists the traditional FII approach in achieving better performance when facing masking effects in the SUT. We also empirically evaluated the efficiency of the test case searching component by comparing it with the random searching based FII approach. The results showed that the ILP-based test case searching method can perform more efficiently. Last, we compared our approach with existing technique for handling masking effects – FDA-CIT [Yilmaz et al. 2014], and observed that our approach achieved a more precise result which can better support debugging.

As for the future work, we plan to do more empirical studies to make our conclusions more general. Our current experiments focus on medium-sized software. We would like to extend our approach to more complicated, large-scaled testing scenarios. Another promising work in the future is to integrate the white-box testing technique into the FII approaches. We believe gaining insight into source code can help figure out the relationships between multiple faults, and hence facilitate the FII approaches obtaining more accurate results. And last, because the extent to which the FII suffers from masking effects varies with different algorithms, combining these different FII approaches would be desired in the future to further improve identifying MFS of multiple faults.

REFERENCES

James Bach and Patrick Schroeder. 2004. Pairwise testing: A best practice that isnt. 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).

James A Jones, James F Bowring, and Mary Jean Harrold. 2007. Debugging in parallel. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 16–26.

Yu Lei, Raghu Kacker, D Richard Kuhn, Vadim Okun, and James Lawrence. 2008. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing. *Software Testing, Verification and Reliability* 18, 3 (2008), 125–148.

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.

Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 11.

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 (CSUR)* 43, 2 (2011), 11.

Changhai Nie, Henry Leung, and Kai-Yuan Cai. 2013. Adaptive combinatorial testing. In *Quality Software (QSIC), 2013 13th International Conference on*. IEEE, 284–287.

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.

Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. 2003. Automated support for classifying software failure reports. In *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 465–475.

Lior Rokach and Oded Maimon. 2005. Top-down induction of decision trees classifiers-a survey. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 35, 4 (2005), 476–487.

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.

Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.

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, Myra B Cohen, and Adam Porter. 2014. Reducing Masking Effects in CombinatorialInteraction Testing: A Feedback DrivenAdaptive Approach. *Software Engineering, IEEE Transactions on* 40, 1 (Jan 2014), 43–66.

Linbin Yu, Feng Duan, Yu Lei, Raghu N Kacker, and D Richard Kuhn. 2015. Constraint handling in combinatorial test generation using forbidden tuples. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*. IEEE, 1–9.

Linbin Yu, Yu Lei, Mehra Nourozborazjany, Raghu N Kacker, and D Rick Kuhn. 2013. An efficient algorithm for constraint handling in combinatorial test generation. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 242–251.

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.

Alice X Zheng, Michael I Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. 2006. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*. ACM, 1105–1112.

Online Appendix to: Identifying minimal failure-causing schemas in the presence of multiple faults

XINTAO NIU and CHANGHAI NIE, State Key Laboratory for Novel Software Technology, Nanjing University
JEFF Y. LEI, Department of Computer Science and Engineering, The University of Texas at Arlington
Hareton Leung and ALVIN CHAN, Department of Computing, Hong Kong Polytechnic University
Xiaoyin Wang, Department of Computer Science, University of Texas at San Antonio

A. THE DETAIL OF THE EXPERIMENTS

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

Table XIX. Comparison between distinguishing failures and regarded as same failure

	Accurate		Super		Sub		Ignore		Irrelevant		Aggregate		TestNum	
Subject	Distin	One	Distin	One	Distin	One	Distin	One	Distin	One	Distin	One	Distin	One
Hsql1	0.67	0.25	0.08	0	0.17	0.58	0.08	0.17	0.08	0.17	0.84	0.50	11.92	8.13
Hsql2	0.83	0.83	0.17	0	0	0.17	0	0	0	0.67	0.92	0.77	7.67	8.67
Hsql3	0.72	0.72	0.17	0	0.06	0.28	0.06	0	0.06	0.67	0.87	0.58	8.61	9.17
Jflex1	0.75	1.00	0.25	0	0	0	0	0	0	5.75	0.88	0.42	6.50	23.50
Jflex2	0.83	0.83	0.17	0	0	0.17	0	0	0	3.83	0.92	0.52	9.00	20.50
Grep1	0.62	0.59	0.20	0	0.06	0.45	0.12	0	0.12	1.85	0.78	0.39	7.09	10.24
Grep2	0.70	0.70	0.20	0	0.05	0.32	0.05	0	0.05	1.43	0.86	0.51	6.48	9.50
Cli1	1.23	1.23	0.08	0	0	0.08	0	0	0	0.77	0.96	0.79	5.77	7.85
Cli2	0.82	0.24	0.18	0	0	0.71	0	0.06	0	0.24	0.94	0.54	6.53	6.41
Joda1	0.91	0.73	0.09	0	0	0.18	0	0.09	0	0.27	0.97	0.75	7.55	7.73
Joda2	0.88	0.88	0.13	0	0	0.13	0	0	0	0.25	0.96	0.81	6.75	7.13
Joda3	0.92	0.92	0.08	0	0	0.08	0	0	0	0.08	0.97	0.92	4.00	4.25
Lang1	0.86	0.71	0.14	0	0	0.29	0	0	0	0.14	0.95	0.79	6.43	6.71
Lang2	0.83	0.83	0.17	0	0	0.17	0	0	0	0.17	0.92	0.83	5.17	5.33
Jsoup1	0.09	0.06	0.12	0	1.15	1.47	0.12	0.11	0.42	1.02	0.49	0.41	8.27	10.11
Jsoup2	0.67	0.33	0.33	0	0	0.67	0	0	0	0.33	0.83	0.58	5.33	5.33

Table XX. Result of our approach

	Accurate			Super			Sub			Ignore			Irrelevant			Aggregate			TestNum		
Subject	Rep	p(R,D)	p(R,O)	Rep	p(R,D)	p(R,O)	Rep	p(R,D)	p(R,O)	Rep	p(R,D)	p(R,O)	Rep	p(R,D)	p(R,O)	Rep	p(R,D)	p(R,O)	Rep	p(R,D)	p(R,O)
Hsql1	0.54	2E-51	4E-62	0.02	7E-45	2E-30	0.17	0.75	1E-62	0.27	4E-50	1E-42	0.27	4E-50	1E-42	0.67	3E-52	6E-52	17.72	6E-69	2E-75
Hsql2	0.83	NaN	NaN	0.17	NaN	0	0	NaN	0	0	NaN	NaN	0	NaN	0	0.92	NaN	0	11.29	2E-69	3E-65
Hsql3	0.67	0	0	0.11	0	0	0.22	0	0	0	0	NaN	0	0	0	0.89	0	0	13.13	2E-68	9E-67
Jflex1	1	0	NaN	0	0	NaN	0	NaN	NaN	0	NaN	NaN	0	NaN	0	1	0	0	9.68	3E-80	1E-98
Jflex2	0.83	NaN	NaN	0.17	NaN	0	0	NaN	0	0	NaN	NaN	0	NaN	0	0.92	NaN	0	13.12	9E-86	4E-93
Grep1	0.76	1E-34	4E-37	0.04	1E-36	2E-20	0.08	6E-08	1E-44	0.12	0.23	3E-30	0.12	0.23	3E-63	0.82	8E-20	1E-48	13.63	6E-51	1E-42
Grep2	0.86	1E-36	1E-36	0.02	0	0	0.07	1E-13	1E-41	0.05	0.56	2E-22	0.05	0.56	7E-65	0.91	1E-25	4E-51	14.29	1E-52	2E-46
Cli1	1.23	NaN	NaN	0.08	NaN	0	0	NaN	0	0	NaN	NaN	0	NaN	0	0.96	NaN	0	6.77	0	0
Cli2	0.82	NaN	0	0.18	NaN	0	0	NaN	0	0	NaN	0	0	NaN	0	0.94	NaN	0	7.53	0	0
Joda1	0.91	NaN	0	0.09	NaN	0	0	NaN	0	0	NaN	0	0	NaN	0	0.97	NaN	0	9.61	7E-60	1E-58
Joda2	1	0	0	0	0	NaN	0	NaN	0	0	NaN	NaN	0	NaN	0	1	0	0	9.19	1E-56	2E-54
Joda3	1	0	0	0	0	NaN	0	NaN	0	0	NaN	NaN	0	NaN	0	1	0	0	5.38	1E-48	4E-46
Lang1	0.86	NaN	0	0.14	NaN	0	0	NaN	0	0	NaN	NaN	0	NaN	0	0.95	NaN	0	9.14	0	0
Lang2	0.83	NaN	NaN	0.17	NaN	0	0	NaN	0	0	NaN	NaN	0	NaN	0	0.92	NaN	0	7.22	6E-49	7E-48
Jsoup1	0.11	2E-15	2E-27	0.11	0	0	1.23	1E-17	1E-30	0.15	1E-09	6E-14	0.97	2E-30	2E-4	0.41	2E-33	3E-4	14.45	2.5E-40	6E-36
Jsoup2	0.67	NaN	0	0.33	NaN	0	0	NaN	0	0	NaN	NaN	0	NaN	0	0.83	NaN	0	11.67	0	0

Table XXI. Comparison between our approach and fd-cit (2-way strength)

	Accurate			Super			Sub			Ignore			Irrelevant			Aggregate			TestNum		
Subject	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value
Hsql1	1.27	1.53	0.17	0.2	3.7	5.3E-10	1.17	0	4.8E-10	1.4	0.97	0.18	1.6	2.53	0	0.45	0.5	0.37	117.47	71.97	2.1E-11
Hsql2	2	0.07	1.8E-27	0.37	4.9	8.6E-27	0	0	NaN	0	0	NaN	0	0.03	0.33	0.94	0.52	4.8E-27	59.63	63.1	0.56
Hsql3	2	1	NaN	0.4	4.77	2.6E-21	1	0	NaN	0	0.33	0.02	0	0.37	0	0.82	0.57	2.3E-15	70.8	58	8.2E-08
Jflex1	2	0	NaN	0	4	NaN	0	0	NaN	0	0	NaN	0	0	NaN	1	0.5	NaN	64.67	74.33	2.1E-18
Jflex2	2	0	NaN	0.33	5.07	1.4E-43	0	0	NaN	0	0	NaN	0	0	NaN	0.94	0.52	7.1E-24	75.5	74.07	0.57
Grep1	2.4	0.7	3.5E-18	0.4	8.97	3E-16	0.63	0	8.7E-08	0.63	0.13	0.02	0.97	1.97	1.4E-4	0.59	0.46	1.7E-05	83.13	100.27	0.04
Grep2	2.63	1	6.1E-14	0.1	8.63	6.4E-15	0.73	0	8.1E-10	0.23	0.07	0.18	0.4	2	1.3E-07	0.72	0.47	2E-10	89.37	140.57	4.6E-4
Cli1	2	1	NaN	0.8	1.7	7.8E-11	0	0	NaN	0.2	0.3	0.38	0	0.43	5.6E-05	0.8	0.55	2.5E-11	31.6	36.57	0.12
Cli2	2.53	1.57	5.8E-07	0.67	2.1	5.4E-4	0	0	NaN	0.03	0.13	0.17	0	0.8	7.4E-06	0.92	0.65	4.6E-09	36.33	59	1.1E-4
Joda1	1.93	1	1.3E-18	0.33	1.63	7.5E-15	0	0	NaN	0	0	NaN	0	0.37	3.0E-4	0.96	0.66	2E-22	63.67	69.67	0
Joda2	2	0.93	3.4E-20	0	4.2	1.7E-23	0	0	NaN	0	0	NaN	0	0	NaN	1	0.72	2.5E-25	67.83	78	0
Joda3	2	1	NaN	0	2.93	1.2E-32	0	0	NaN	0	0	NaN	0	0.13	0.04	1	0.73	4.9E-21	58.63	79.27	1.8E-4
Lang1	1.97	0.67	2.7E-16	0.37	2.67	1.2E-08	0	0	NaN	0	0	NaN	0	0.37	3.1E-4	0.96	0.61	2.8E-22	46.1	58.97	0
Lang2	2	0.07	1.8E-27	0.17	5.13	5.9E-25	0	0	NaN	0	0	NaN	0	0	NaN	0.97	0.52	9.1E-35	43.97	60.1	3.3E-4
Jsoup1	0.3	1.17	2.2E-07	0.2	3	1.4E-10	3.73	0.5	5.8E-21	0.7	1.6	0.01	1.9	1.97	0.87	0.4	0.42	0.52	53.53	132.67	5.2E-05
Jsoup2	2	2	NaN	1	1	NaN	0	0	NaN	0	0	NaN	0	0	NaN	0.83	0.83	NaN	68.47	79.67	0.03

Table XXII. Comparison between our approach and fd-cit (3-way strength)

	Accurate			Super			Sub			Ignore			Irrelevant			Aggregate			TestNum		
Subject	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value
Hsql1	1.33	0.87	0	0.13	8.87	1.8E-12	1.23	0	7.3E-11	1.17	0	7.6E-06	2.43	0.93	8.7E-05	0.44	0.66	3.5E-07	205.2	182.13	7.4E-4
Hsql2	2	0	NaN	0.23	5	4.1E-32	0	0	NaN	0	0	NaN	0	0	NaN	0.96	0.53	1.3E-24	114.9	129.8	2.8E-14
Hsql3	2	1	NaN	0.3	5.83	3.5E-48	1	0	NaN	0	0	NaN	0	0	NaN	0.82	0.63	1.3E-36	122.27	129.87	1.5E-4
Jflex1	2	0	NaN	0	4	NaN	0	0	NaN	0	0	NaN	0	0	NaN	1	0.5	NaN	145.77	190.53	6.2E-52
Jflex2	2	0	NaN	0.43	5	1.3E-29	0	0	NaN	0	0	NaN	0	0	NaN	0.93	0.53	1.1E-21	161.23	187.47	3.3E-16
Grep1	2.3	0.93	8.3E-16	0.33	12.9	3.8E-22	0.67	0	2.1E-08	0.77	0	0	1.47	1.17	0.26	0.53	0.48	0.08	123.57	178.4	0
Grep2	2.9	1	1.4E-12	0.1	13.53	7.9E-23	0.63	0	8.7E-08	0.07	0	0.33	0.53	0.77	0.25	0.76	0.5	1.5E-10	150.23	255.93	4.5E-05
Cli1	2	1	NaN	1	2	NaN	0	0	NaN	0	0	NaN	0	0	NaN	0.83	0.67	NaN	48.63	77.57	0
Cli2	2.93	1.97	5.6E-23	0.5	1.9	7.4E-12	0	0	NaN	0	0	NaN	0	0.2	0.01	0.96	0.74	5.7E-25	52.87	114.8	5.5E-4
Joda1	2	1	NaN	0.3	2	1.6E-18	0	0	NaN	0	0	NaN	0	0	NaN	0.97	0.72	2.8E-22	132.7	161.23	1.9E-20
Joda2	2	1	NaN	0	4	NaN	0	0	NaN	0	0	NaN	0	0	NaN	1	0.73	NaN	147.07	184.77	0
Joda3	2	1	NaN	0	3	NaN	0	0	NaN	0	0	NaN	0	0	NaN	1	0.75	NaN	135.3	158.4	0.01
Lang1	2	0.77	1E-15	0.27	2.8	8.4E-10	0	0	NaN	0	0	NaN	0	0.03	0.33	0.97	0.68	7.7E-22	76.77	128.87	1.7E-4
Lang2	2	0	NaN	0.3	5	6.1E-31	0	0	NaN	0	0	NaN	0	0	NaN	0.95	0.53	2.7E-23	86.9	108.7	8.8E-07
Jsoup1	0.27	1.87	1.5E-21	0.33	4.97	1.1E-23	3.57	0	5E-19	0.7	1.03	0.15	1.63	1.83	0.58	0.42	0.49	0.01	81.4	172.57	6.6E-4
Jsoup2	2	2	NaN	1	1	NaN	0	0	NaN	0	0	NaN	0	0	NaN	0.83	0.83	NaN	103.9	118.5	2.3E-19

Table XXIII. Comparison between our approach and fd-cit (4-way strength)

	Accurate			Super			Sub			Ignore			Irrelevant			Aggregate			TestNum		
Subject	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value	Rep	Fd-cit	p-value
Hsql1	1.43	0.9	3.3E-4	0.13	8.83	5.3E-18	1.6	0	1.3E-15	0.6	0	9.6E-4	2.5	0	2.6E-10	0.49	0.7	1.3E-08	387.2	418.83	2.8E-05
Hsql2	2	0	NaN	0.2	5	6.7E-33	0	0	NaN	0	0	NaN	0	0	NaN	0.97	0.53	1.9E-25	239.23	297.13	6.4E-34
Hsql3	2	1	NaN	0.17	6	3.1E-36	1	0	NaN	0	0	NaN	0	0	NaN	0.83	0.64	4.9E-36	248.03	297.67	3.1E-22
Jflex1	2	0	NaN	0	4	NaN	0	0	NaN	0	0	NaN	0	0	NaN	1	0.5	NaN	330.8	451.47	5.2E-68
Jflex2	2	0	NaN	0.33	5	1.7E-30	0	0	NaN	0	0	NaN	0	0	NaN	0.94	0.53	8.7E-23	357.8	457.73	1.2E-44
Grep1	2.17	1	2.5E-14	0.43	22	8.0E-42	0.8	0	1.1E-11	0.3	0	0.06	0.93	0	1.4E-4	0.61	0.45	4.6E-07	178.53	255.63	0
Grep2	2.63	1	6.3E-12	0.07	19.6	2.1E-41	0.77	0	1.1E-10	0.03	0	0.33	0.3	0	0.03	0.77	0.47	3.2E-14	244.33	303.53	0.01
Cli1	2	1	NaN	1	2	NaN	0	0	NaN	0	0	NaN	0	0	NaN	0.83	0.67	NaN	76.93	95	1.9E-12
Cli2	3	2	NaN	0.57	2	1.2E-15	0	0	NaN	0	0	NaN	0	0	NaN	0.95	0.77	8.1E-21	80.1	151.63	1.2E-4
Joda1	2	1	NaN	0.43	2	1.2E-16	0	0	NaN	0	0	NaN	0	0	NaN	0.95	0.72	6.9E-20	266.63	355.4	1.3E-39
Joda2	2	1	NaN	0	4	NaN	0	0	NaN	0	0	NaN	0	0	NaN	1	0.73	NaN	299.6	370.87	6.4E-42
Joda3	2	1	NaN	0	3	NaN	0	0	NaN	0	0	NaN	0	0	NaN	1	0.75	NaN	278.17	338.1	0
Lang1	2	1	NaN	0.37	2	1.9E-17	0	0	NaN	0	0	NaN	0	0	NaN	0.96	0.72	1.3E-20	138.83	191.87	2.1E-38
Lang2	2	0	NaN	0.3	5	6.1E-31	0	0	NaN	0	0	NaN	0	0	NaN	0.95	0.53	2.7E-23	163.03	210.73	6.1E-09
Jsoup1	0.4	1.93	9.8E-19	0.33	6.6	1.2E-43	3.8	0	6.5E-20	0.3	0	0.02	1.8	0.7	0	0.44	0.6	6.2E-11	133.1	226.6	0.03
Jsoup2	2	2	NaN	1	1	NaN	0	0	NaN	0	0	NaN	0	0	NaN	0.83	0.83	NaN	172.37	243.63	1.5E-4