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

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

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 failure in the System Under Test (SUT). However, we argue that multiple failures are the more common testing scenario, under which masking effects may be triggered so that some failures cannot be observed. The traditional MFS theory, as well as the related identifying algorithms, lack a mechanism to handle such effects; hence, they may incorrectly isolate the MFS in the SUT. To address this problem, we propose a new MFS model that takes into account multiple failures. We first formally analyse the impact of the multiple failures on existing MFS identifying algorithms, especially in situations where masking effects are triggered by multiple failures. We then develop an approach that can assist traditional algorithms to better handle multiple failures scenario. Empirical studies were conducted using several kinds of open-source software, which showed that multiple failures 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 failures. *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2010), 35 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 failure [Song et al. 2012]. While exhaustive testing achieves maximal test coverage, it

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

is impractical and uneconomical. One remedy for this problem is combinatorial testing, 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 code scope that needed for inspection [Ghandehari et al. 2012]. However, information from a covering array sometimes is not sufficient to identify the location and number of the failure-inducing interactions [Colbourn and McClary 2008]. Thus, additional information is 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 five two-way suspicious interactions that may be responsible for this failure. They are respectively (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 SUT, based on which additional test cases were generated to identify them. Other approaches to identify the failure-inducing interactions in 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 failure, under which the outcomes of test cases can be simply categorized into fail or pass. However, in this paper, we argue that SUT with multiple distinguished failures is the more common testing scenario in practice, and moreover, this affects the effectiveness of Failure-inducing Interactions Identifying (FII) approaches. One main impact of multiple failures on FII approaches is 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 example. We noticed that there are two different failures reported in the bug tracker system. The first [1] claims that Grep incorrectly matches unicode patterns with

[1] http://savannah.gnu.org/bugs/?29537

‘\<\>’, while the second [2] 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 will prevent test cases from executing 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 effects 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 multiple failures as one 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 is 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 re-generated test cases. In this way, FII approaches can properly work with little interference from the negative masking effects.

The key to our strategy is to search for a test case that does not trigger *unexpected* failures, i.e., failures different from the one under analysis. To guide the search process, a natural idea is to take some characteristics from the existing test cases, and make the characteristics of the newly searched test case as disparate from the test cases which triggered unexpected failures as possible. To reach this target, we define the *suspiciousness* between a factor and the failure. The higher the *suspiciousness* a factor is related to a particular failure, the greater the likelihood that the factor will trigger this failure. We then use the integer linear programming (ILP) technique to find a test case which has the least *suspiciousness* with unexpected failures.

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 two open-source software systems found in the developers’ forum in the Source-Forge community. Through studying their bug reports in the bug tracker system as well as their user’s 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 empirically studied the performance of the important component of our strategy – searching satisfied test cases. To conduct this study, we compared our approach with the augmented FII approach by randomly searching satisfied test cases. We finally compared our approach with the only existing masking handling technique – FDA-CIT [Yilmaz et al. 2014]. Our studies showed that our replacing strategy as well as the searching test case component achieved a better performance than the traditional approaches when the subject suffered multiple failures, especially when these failures can import masking effects.

The main contributions of this paper are:

— We formally analysed the relationships between failure-inducing interactions and test sets. (Section 3)

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

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

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

  return x+y;
  }
```

Fig. 1. A simple program *foo* with four input parameters

— We studied the impact of the masking effects caused by multiple failures on FII approaches. (Section 4)
— We proposed an efficient test case replacement strategy to alleviate the impact of these effects (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 SUT with masking effects. (Section 6)

2. MOTIVATING EXAMPLE

We constructed a small example to illustrate the motivation of 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).

Traditional FII algorithms do not consider the code detail; instead, they apply black-box testing, i.e., feed inputs to the programs and execute them to observe the result. The basic justification behind these approaches is that the failure-inducing interactions for a particular failure can only appear in those test cases that trigger this failure. Traditional 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 these FII approaches can achieve. Next, we will show how exhaustive testing works to identify the failure-inducing interactions for the program.

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 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.

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

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

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

test cases contain the interaction (-,-,4,5), but they did not trigger Ex 2; instead, Ex 1 was triggered.

Now consider the source code of *foo*. We can find that if Ex 1 is triggered, it will stop executing the remaining code and report the exception result. In other word, 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.

Unless we fix the code that triggers Ex 1 and re-execute all the test cases, we cannot validate the supposition that *test case 6* and *test case 18* should trigger Ex 2 when they did not trigger Ex 1. 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 in CT

Assume that the behaviour of 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, some of them (Definitions 3.1, 3.3, 3.4) were originally defined in [Nie and Leung 2011b].

Definition 3.1. A *test case* of SUT is an array of *k* values, one for each parameter of the SUT, which is denoted as a *k*-tuple (v_1, v_2,...,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. In this paper, we focus on studying the impact of multiple *failures* on failure-inducing interactions identification. 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 the most common assumption of CT fault diagnosis. 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.

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_{k_1},...,v_{k_t},...) 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 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 all test cases contain a schema c, and trigger a 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.

Some notations used later are listed below for convenient reference:

— k : The number of parameters that influence the SUT.
— V_i : The set of discrete values that the ith parameter of 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 $\prod_{i=1}^{i<=k}|V_i|$. Note that some test cases may be invalid if there exists constraints among the parameters.
— T : A set of test cases. (Similarly for $T_i,\ T_j, ...$)
— $\bar{T}$: 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 failures contained in the SUT.
— F_m : The mth failure in the SUT ($1 \leq m \leq L$); for different failures, we can differentiate them based on their exception traces or other information.
— 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 $\mathcal{T}(c) \subseteq T_{F_m}$.
— $\mathcal{I}(t)$: 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., $\mathcal{T}(c_2) \subseteq \mathcal{T}(c_1)$.*

PROOF. $\forall t \in \mathcal{T}(c_2)$, t contains c_2. As t is a k-degree schema, then it has $c_2 \prec t$. As $c_1 \prec c_2$, then $c_1 \prec t$, indicating that t contains c_1. Therefore, $t \in \mathcal{T}(c_1)$. So it follows that $\mathcal{T}(c_2) \subseteq \mathcal{T}(c_1)$. □

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 (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 (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} denotes the set of all the test cases triggering 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 $\mathcal{T}(c) \subseteq T$, $c \in \mathcal{S}(T)$.*

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, -).

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)

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 relationships 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 $c_1 \prec c_2$.*

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 $c_1 \prec c_2$.*

Proposition 3.11 is exactly the antithesis of Proposition 3.10. 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 also 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 set

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 meaningful, they clarify the relationships between the minimal schemas of two different test sets (Propositions 3.11 and 3.10), which is the key to explain the impacts of masking effects in the following section.

4. MASKING EFFECT

As discussed before, $\mathcal{C}(T_{F_m})$ is the MFS set of failure F_m in theory. When considering the masking effects between multiple failures, however, this formula 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 another failure was triggered ahead of it that prevents t from being normally checked.

Taking the masking effects into account, when identifying the MFS of a specific failure F_m, we should 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 $\mathcal{C}(T_{F_m} \bigcup T_{mask(F_m)})$.

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 the 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 failures problem, i.e., *regarded as one failure* and *distinguishing failures*. The former strategy treats all types of failures as one failure–*fail*, and others as *pass*, 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 one 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 all the 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 s-trategy 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 failures in column $T_{mask(F_1)}$ and T_{non_mask}, 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 we listed them in the column 'actual MFS of *failure 1*'. However, when we use the *regarded as one 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).

Table VIII. masking effects for exhaustive testing

T_{F_1}	$T_{mask(F_1)}$	T_{non_mask}
(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* $\mathcal{C}(T_{F_1} \bigcup T_{mask(F_1)})$	regarded as one failure $\mathcal{C}(T_{F_1} \bigcup T_{mask(F_1)} \bigcup T_{non\_mask})$	distinguishing failures $\mathcal{C}(T_{F_1})$
(1, 1, -, -)	(-, -, 0, 0)	(1, 1, -, 1)
(-, 1, 1, 1)	(1, 1, -, -)	(1, 1, 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 SUT with multiple failures.

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 3.11, some schemas obtained by this strategy may be the super-schema of some actual MFS. Moreover, some MFS may be irrelevant to the schemas obtained by this strategy, 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 MF-S 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.

We offer 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 pass of t_1 and t_3 indicates 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_3) 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,-).

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	-)	

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 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_3 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 one 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.

Table XII. OFOT with two strategies

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

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 one 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. TEST CASE REPLACING STRATEGY

The main reason that the FII approach fails to work properly is that it cannot determine $T_{mask(F_m)}$. In other word, 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 XIII. Inferred test cases for two strategies

regarded as one 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 it can reduce the probability that the expected failure may be masked by other 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 the performance unless we fix other 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 regenerated test cases should either pass in the execution or trigger F_m. The replacement must satisfy the condition that the newly generated ones will not negatively influence the original identifying process.

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, if 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, the 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 are dynamically changed, depending on the outcome of the execution of last generated test case.

This replacement process may need to be executed multiple times for one fixed part as it may not always possible to find a test case that coincidentally satisfied our requirement. One replacement method is randomly choosing test cases until the satisfied test case is found. While this method may be simple and straightforward, however, it also may require tries. So to handle this problem and reduce the cost, we propose a replacement approach by computing the *suspiciousness* of the test case with the other failures, and then selecting the test case from a group of candidate test cases that has the least *suspiciousness* with other failures.

To explain the *suspiciousness* notion, we first introduce the *suspiciousness* between a parameter value o and a particular failure. We use $all(o)$ to represent the number of executed test cases that contain this parameter value, and $f_i(o)$ to indicate the number of test cases that trigger the failure $F_i, 1 \leq i \leq L$ and contain this parameter value. Then, the *suspiciousness* between a parameter value and a particular failure, i.e., $Sp(o, F_i)$, is $\frac{f_i(o)}{all(o)+1}$. This heuristic formula is based on the idea that if a parameter value frequently appears in the test cases that trigger a particular failure, then it is more likely to be the inducing factor that triggers this failure. We add 1 in the denominator for two reasons: (1) avoid division by zero when the parameter value has never appeared before, (2) reduce the bias when a parameter value rarely appears in the test set but coincidentally appears in a failing test case with a particular failure.

Table XIV gives an example to compute the suspiciousness between parameter values and particular failures. The left part of this table gives 4 executed test cases with their outcomes, in which failures F_1 and F_2 are triggered. The left part shows the suspiciousness between each parameter value and these two failures. Specifically, consider the parameter P_1 taking value 0. There are three test cases t_1, t_2, and t_4 that contain this parameter value, and only t_2 triggered F_1. Hence, $Sp(P_1 = 0, F_1)$ = $\frac{f_1(o)}{all(o)+1}$ = $\frac{1}{3+1}$ = 1/4.

Table XIV. Suspiciousness example

ID	Test cases executed	Outcomes
t_1	(0, 0, 0)	$Pass$
t_2	(0, 1, 1)	F_1
t_3	(1, 1, 0)	F_2
t_4	(0, 1, 1)	F_2

Suspiciousness for $F1$			
	P_1	P_2	P_3
0	1/4	0	0
1	0	1/4	1/3
Suspiciousness for $F2$			
	P_1	P_2	P_3
0	1/4	0	1/3
1	1/2	1/2	1/3

With the *suspiciousness* associated with a parameter value, we then define the *suspiciousness* between a test case t and a particular failure F_i as:

$$Sp(t, F_i) = \frac{1}{k} \sum_{o \in t} Sp(o, F_i) \qquad \textbf{(EQ1)}$$

where k is the number of parameters in t, and o is the specific parameter value in t. The *suspiciousness* between a test case and a failure is actually the average *suspiciousness* between each parameter value in the test case and this failure. For example, considering an unexecuted test case (1, 0, 0). With the suspiciousness of all the parameter values listed Table XIV, $Sp((1, 0, 0), F_1)$ = $1/3 \times (0 + 0 + 0) = 0$ and $Sp((1, 0, 0), F_2)$ = $1/3 \times (1/2 + 0 + 1/3) = 5/18$.

For a selected test case, we want its ability to trigger other failures to be as small as possible, such that the masking effects can be alleviated. In practice, the *suspiciousness* varies between test case and different failures. As a result, we cannot always find a test case that, for any failure, the *suspiciousness* between this test case and that failure is the least.

Table XV illustrates such an scenario for SUT(2^4). Suppose the FII approach is analysing MFS for failure F_1, and needs to replace test case (0,0,0,0) with fixed part (0, -, -, -) that triggers other failure, i.e., F_2 or F_3. This table lists five candidate test cases, with their suspiciousness with F_2 and F_3 given in the corresponding columns. It is obvious that t_1 has the least suspiciousness with failure F_3 and t_2 has the least suspiciousness with failure F_2. These two test cases, however, should not be selected for their suspiciousness with another failure is too high. Instead, t_3 is a good choice as both its suspiciousness with F_2 and F_3 is not high (The higher one is just 0.4).

With this in mind, we have to settle for a test case, such that the maximal possible failure (except for the one that is currently analysed) it can trigger should be the least likely to be triggered when compared with that of other test cases. In other word, we need to find a test case, so that the maximal *suspiciousness* between this test case and the corresponding failure is minimal. Formally, we should choose a test case t, s.t.,

Table XV. Select minimal maximal suspiciousness

ID	Candidate test cases	$Sp(t, F_2)$	$Sp(t, F_3)$	$Max\ Sp(t, F_m), m = 2, 3$
t_1	(0, 0, 0, 1)	0.7	0.2	$Sp(t_1, F_2) : 0.7$
t_2	(0, 0, 1, 0)	0.2	0.6	$Sp(t_2, F_3) : 0.6$
$\mathbf{t_3}$	**(0, 0, 1, 1)**	**0.4**	**0.3**	$\mathbf{Sp(t_3, F_2) : 0.4}$
t_4	(0, 1, 0, 0)	0.3	0.5	$Sp(t_4, F_3) : 0.5$
t_5	(0, 1, 0, 1)	0.5	0.3	$Sp(t_5, F_2) : 0.5$

$$\min_{t \in R} \max_{1 \leq i \leq L \& i \neq m} Sp(t, F_i) \tag{EQ2}$$

where L is the number of failures, and m is the current analysed failure. R is the set of all possible test cases that contain the $fixed$ part except those that have been tested. As $fixed$ part is a set of parameter values which can be deemed as a schema, then obviously $R = \mathcal{T}(fixed) \backslash T_{executed}$, where $\mathcal{T}(fixed)$ is all the test cases that contain this fixed part and $T_{executed}$ represents those executed test cases. Additional test cases need to be selected in set R, so that FII approaches can work properly.

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, the suspiciousness matrix Sp
Output: t_{new} the regenerate test case

1 **while** ***not*** *MeetEndCriteria()* **do**
2 $\quad t_{new} \leftarrow \min_{t \in R} \max_{1 \leq i \leq L \& i \neq m} Sp(t, F_i)$;
3 $\quad result \leftarrow execute(t_{new})$;
4 $\quad updatet\_SP(t_{new})$;
5 $\quad$ **if** $result == PASS\ or\ result == F_m$ **then**
6 $\quad\quad$ **return** t_{new};
7 $\quad$ **else**
8 $\quad\quad$ continue;
9 $\quad$ **end**
10 **end**
11 **return** *null*

The inputs to this algorithm consist of the failure F_m under analysis, the candidate test cases $R = \mathcal{T}(fixed) \backslash T_{executed}$ and the suspiciousness matrix Sp, which records the suspiciousness between each factor o and each failure F_i, i.e., $Sp(o, F_i)$ ($1 \leq i \leq L$). 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 three parts:

The first part (lines 2 - 3) generates and executes a new test case which is supposed to be least likely to trigger failures different from F_m. The new test case is generated according to EQ2. In our implementation, we use the solver introduced in [Berkelaar et al. 2004], which is a mixed Integer Linear Programming (MILP) solver suitable for satisfaction and optimization problems.

The second part (line 4) updates the suspiciousness matrix (Sp) for each parameter value that is involved in this newly generated test case (line 4). Specifically, for a particular parameter value o, the number of executed test cases that contain o, i.e., $all(o)$, increases by 1. Additionally, if this test case triggers failure F_i ($1 \leq i \leq L$) , then the number of test cases that contain o and trigger failure F_i, i.e., $f_i(o)$, increases by 1. At last, the suspiciousness value will be re-computed according to formula $Sp(o, F_i) = \frac{f_i(o)}{all(o)+1}$ $(1 \leq i \leq L)$.

The last 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).

Note that this algorithm has another exit, besides finding an expected test case (line 6), which is when the function *MeetEndCriteria()* returns *true* (line 1). We did not explicitly show function *MeetEndCriteria()*, because this is dependent on the computing resource and the desired accuracy. In detail, if we want to get a high quality result and have enough computing resource, it is desirable to try many times to get the expected test case; otherwise, a relatively small number of attempts is recommended.

5.2. A case study using the replacement strategy

Suppose we have to test a system with eight parameters, each of which has three options, i.e., SUT(3^8). When we execute the test case t_0 – (0, 0, 0, 0, 0, 0, 0, 0), a failure–$e1$ is triggered. Furthermore, there are two more potential failures, $e2$ and $e3$, that may be triggered during the testing and they will mask the desired failure $e1$. Next, we will use FIC_BS [Zhang and Zhang 2011] with replacement strategy to identify the MFS of $e1$. The process is shown in Figure 2. In this figure, there are two main columns. The left main column indicates the executed test cases during testing as well as the executed results, with each executed test case corresponding to a specific label, $t_1 - t_8$, at the left. The underline part for each test case is the $fixed$ part according to FIC_BS [Zhang and Zhang 2011]. The right main column lists the suspiciousness matrix when a test case triggers $e2$ or $e3$. The executed test case, shown in bold, indicates the one that triggers the other failure and should be replaced in the next iteration.

The completed MFS identifying process listed in Figure 2 works as follows: firstly the original FII approach determines which $fixed$ part needed to be test in each iteration. Then an extra test case will be generated to fill in the remaining part. After executing the extra test case, if the result of the execution is normal, i.e., did not trigger any unexpected failure (e_2, e_3), then the original FII process will continue until the MFS is identified. Otherwise, the replacement strategy starts when an unexpected failure is triggered. The replacement process will mutate the parameter values that are not in the $fixed$ part according to EQ2. After the replacement process, the control for the MFS identifying process will be passed back to the original FII approach. Next we will specifically explain how the replacement works with an example in Figure 2.

From Figure 2, for the test case that triggered $e2$ – (2, 1, 1, 1, 0, 0, 0, 0) (in this case, the fixed part of the test case is (-, -, -, -, 0, 0, 0, 0), in which the last four parameter values are the same as the original test case t_0), we generate the related matrix at left. Each element in this matrix is computed according to EQ1. All the suspiciousness with $e3$ is labeled with a short slash as there is no test case triggering this failure in this iteration. After this matrix has been determined, we can obtain the optimal test case with the ILP solver, which is t_1'–(1, 2, 2, 2, 0, 0, 0, 0), with its suspiciousness 0.167, which is smaller than all other test cases.

This replacement process is started each time a new test case that triggered another failure until we finally get the MFS. Sometimes we could not find a satisfied replacing test case in just one trial like t_1 to t_1'. When this happened, we needed to repeat searching the proper test case. For example, for t_4 which triggered $e3$, we tried three times–t_4', t_4'', t_4''' to finally get a satisfied t_4''' which passes the testing. Note that the *suspicious-*

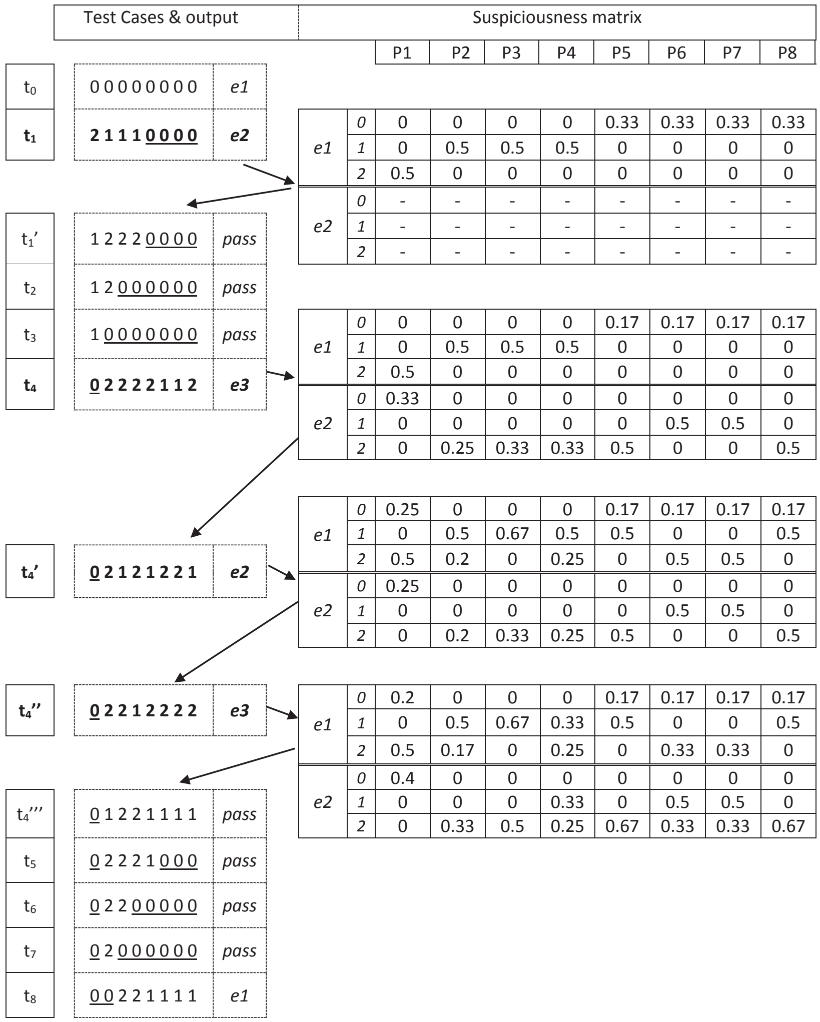


Fig. 2. A case study using our approach

ness matrix continues to change as the test case is generated and executed so that we can adaptively find an optimal one.

With the replacement test cases, the FII approach can properly work. At last, the MFS identified for failure $e1$ is (0,0, -, -, -, -, -, -) (Test cases passed when we mutated the first two parameter values of the original failing cases). In this example, test cases that trigger $e2$ and $e3$ do not contribute to the MFS identification. In real-world scenario, however, it is appealing to iteratively start MFS identification for $e2$ and $e3$ based on these test cases.

6. 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 failures?
Q2: How well does our approach perform compared to traditional approaches?
Q3: Is the ILP-based test case searching technique efficient compared to the random selection?
Q4: Compared to another masking effects handling approach FDA-CIT [Yilmaz et al. 2014], does our approach have any advantages ?

6.1. The existence and characteristics of masking effects

In the first study, we surveyed two kinds of open-source software systems to gain an insight into the existence of multiple failures and their effects. The software under study were HSQLDB and JFlex. The first is a database management software written in pure Java and the second is a lexical analyser generator. The reason that we chose these two systems is because they both 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 uncommented code, number of classes in the project, and the bug's id [3] for each of the software.

Table XVI. Software under survey

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

6.1.1. 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 later regarded as the "prior MFS".

We further built the testing scenario for each version of the software listed in Table XVI. The testing scenario 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 models information is shown in Table XVII for HSQLDB and JFLex, 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

[3] http://sourceforge.net/p/hsqldb/bugs
http://sourceforge.net/p/jflex/bugs

9 parameters that can take 2 values, 2 parameters 3 values, and only one parameter 4 values.

Table XVII. Input models of HSQLDB and JFlex

HSQLDB

	common options	values
	Server Type	3
	existed form	2
	resultSetTypes	3
	resultSetConcurrencys	2
	resultSetHoldabilitys	2
	StatementType	2
	sql.enforce_strict_size	2
	sql.enforce_names	2
	sql.enforce_refs	2
versions	**specific options**	**values**
2.0rc8	more	2
	placeHolder	2
	cursorAction	4
2.2.5	multiple	3
	placeHolder	2
2.2.9	duplicate	3
	defailure_commit	2
versions	**Config space**	
2.0rc8	$2^9 \times 3^2 \times 4^1$	
2.2.5	$2^8 \times 3^3$	
2.2.9	$2^8 \times 3^3$	

JFlex

	common options	values
	generation	3
	charset	4
	public	2
	apiprivate	2
	cup	2
	caseless	2
	char	2
	line	2
	column	2
	notunix	2
	yyeof	2
versions	**specific options**	**values**
1.4.1	hasReturn	3
	normal	2
1.4.2	lookAhead	3
	type	2
	standalone	2
versions	**Config space**	
1.4.1	$2^{10} \times 3^2 \times 4^1$	
1.4.2	$2^{11} \times 3^2 \times 4^1$	

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 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.

6.1.2. Results and discussion. Table XVIII 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, and column "masking" indicates the number of masked test cases. The percentage in the parentheses 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 XVIII, test cases with masking effects do exist, i.e., test cases containing MFS did not trigger the corresponding bug. In fact, there are about 768 out of 4608 test cases (16.7%) in hsqldb with 2rc8 version. This rate is about 16.7%, 50%, 25%, and 16.7%, respectively, for the remaining software versions.

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

Table XVIII. Number of failures and their masking effects

software	versions	all tests	failure	masking
HSQLDB	2cr8	18432	4608	768 (16.7%)
-	2.2.5	6912	3456	576 (16.7%)
-	2.2.9	6912	3456	1728 (50%)
JFlex	1.4.1	36864	24576	6144 (25%)
-	1.4.2	73728	36864	6144 (16.7%)

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 can be either MFS or not. 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 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 the test cases which contain the MFS but do not 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.

6.2. 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 five versions of software in Table XVI used as test objects are far from the requirement for a general evaluation. However, to construct many real testing objects is time-consuming as we must carefully study the detail of that software as well as the bug tracker report. To compromise, we synthesized 10 more testing objects. These synthesized objects are ten small programs which can directly return outputs when executed with given inputs. To make the synthetic objects as similar as possible to the real software, we firstly analysed the characterizations, such as the number of parameters, the number of failures, and the possible masking effects, of the real software. We observed that the number of parameters of the SUT ranged from 8 to 30, the number of different failures in the SUT ranged from 2 to 4, and the number of MFS of a failure ranged from 1 to 2, in which the degree of the MFS ranged from 1 to 6. Then for each characterization, we randomly selected one value in the corresponding range and assigned it to the input model by adjusting the relationships between the inputs and outputs of these programs.

Table XIX 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 five software as $H2cr8$, $H2.2.5$, $H2.2.9$, $J1.4.1$, $J1.4.2$, 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 will mask the right failure, i.e., if the left failure is triggered, then the right failure 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. By using the formula $e_1 > e_2 > e_3$, we indicate 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 the masking effects can happened 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.

6.2.1. 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 ap-

Table XIX. The testing models used in the case study

Object	Model	Failures	MFS of each failure
H2cr8	$2^9 \times 3^2 \times 4^1$	$e_1 > e_2 > e_3$	$(5_1, 6_0, 7_0)_{e_1}, (5_1, 8_2, 9_2)_{e_2}, (5_1, 8_2, 9_1)_{e_2}, (5_1, 8_3, 9_2)_{e_3}, (5_1, 8_3, 9_1)_{e_3}$
H2.2.5	$2^8 \times 3^3$	$e_1 > e_2$	$(6_1, 7_0)_{e_1}, (5_2)_{e_2}$
H2.2.9	$2^8 \times 3^3$	$e_1 > e_2 > e_3$	$(6_0)_{e_1}, (0_1, 5_1, 7_0)_{e_2}, (0_0, 5_1, 7_0)_{e_2}, (5_1, 7_0)_{e_3}$
J1.4.1	$2^{10} \times 3^2 \times 4^1$	$e_1 > e_2$	$(0_0)_{e_1}, (1_0)_{e_2}$
J1.4.2	$2^{11} \times 3^2 \times 4^1$	$e_1 > e_2$	$(1_0, 2_1)_{e_1}, (0_1)_{e_2}$
syn1	$2^5 \times 3^3 \times 4^1$	$e_1 > e_2$	$(2_1, 3_0)_{e_1}, (1_1, 2_1)_{e_2}, (1_0, 3_0)_{e_2}$
syn2	$2^6 \times 3^2 \times 4^1$	$e_1 > e_2 > e_3$	$(4_1, 6_0, 7_1, 8_0)_{e_1}, (1_1, 3_1, 5_1)_{e_2}, (2_0, 3_1, 6_0)_{e_3}$
syn3	$2^5 \times 3^3$	$e_1 > e_2 > e_3$	$(2_1, 3_0)_{e_1}, (1_0)_{e_2}, (4_1)_{e_2}, (6_0, 7_0)_{e_3}$
syn4	$2^7 \times 3^2 \times 4^1$	$e_1 > e_2 > e_3$	$(0_1, 2_1, 5_0, 6_1)_{e_1}, (2_1, 4_0)_{e_2}, (6_1, 7_0)_{e_2}, (3_0, 4_0, 5_0)_{e_3}$
syn5	$2^4 \times 3^3 \times 4^2$	$e_1 > e_2$	$(0_0, 1_1, 3_0, 6_1, 8_0)_{e_1}, (2_0, 3_0, 4_1)_{e_2}$
syn6	$2^9 \times 3^2$	$e_1 > e_2 > e_3 > e_4$	$(2_0, 7_1, 8_1)_{e_1}, (3_1, 5_1)_{e_2}, (4_0)_{e_2}, (3_1, 6_0, 7_1)_{e_3}, (3_1, 7_1, 8_0)_{e_4}$
syn7	$2^{10} \times 3^1 \times 4^1$	$e_1 > e_2 > e_3$	$(3_1, 4_0, 5_0)_{e_1}, (2_0, 4_0, 7_1, 9_0)_{e_2}, (6_1, 10_0, 11_1)_{e_3}$
syn8	$2^{11} \times 3^1 \times 4^1$	$e_1 > e_2$	$(1_0, 3_1, 4_0, 7_1, 9_0, 12_1)_{e_1}, (0_0, 2_1, 3_1, 7_1, 10_0, 11_1)_{e_2}$
syn9	$2^4 \times 4^3$	$e_1 > e_2$	$(3_1, 5_0)_{e_1}, (5_0, 6_1)_{e_2}$
syn10	$2^7 \times 3^3 \times 4^1$	$e_1 > e_2$	$(0_1, 3_0, 4_1, 7_0)_{e_1}, (2_0, 3_0, 5_1)_{e_2}, (2_0, 3_0, 5_0)_{e_2}$

proach 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 failures 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 one 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 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.
(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 to identify as many actual MFS as possible is the target for every FII approach. 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 some 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.

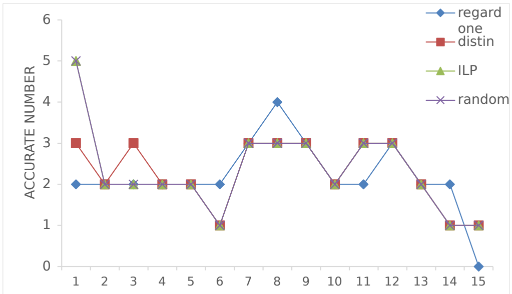
The *related* function is the summation of similarity of all the super or sub schemas with their corresponding MFS.

6.2.2. Results and discussion. Figure 3 depicts the results of the second case study. There are seven sub-figures in this figure, i.e., Figure 3(a) to Figure 3(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.

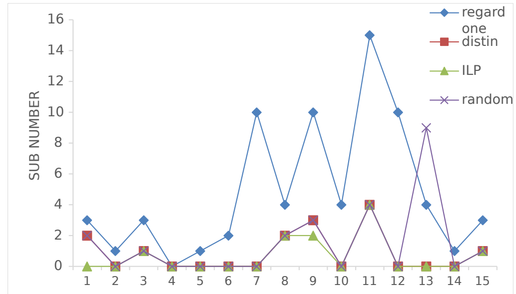
For each sub-figure, there are four polygonal lines, each of which shows the results for one of the four strategies: *regarded as one failure*, *distinguishing failures*, *replacement strategy based on ILP searching*, *replacement strategy based on random searching* (The last one will be discussed in 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 3(a), the point marked with '♦' at (1,2) indicates that the approach using *regarded as one failure* strategy identified 2 accurate MFS in the testing object–HSQLDB 2cr8.

The raw data for this experiment can be found in Table XX of the Appendix. Note that all the data except for the metric *ignored number* are based on all failing test cases for each testing object, i.e., we got the data by comparing the union of the schemas identified in each of the failing test cases to the prior actual MFS. As for metric *ignored number*, however, we found that if we merged all the schemas identified in each failing test case, there is no MFS ignored. We therefore use the average score of *ignored number* for each failing test case, which can be seen in the parentheses in Column *ignore* of Table XX.

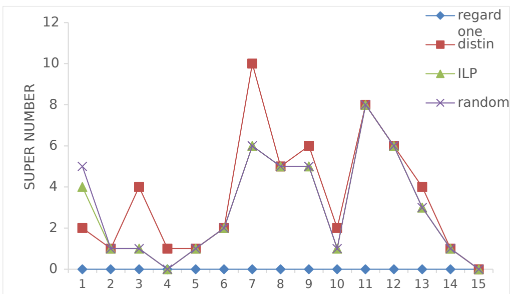
Accurate number: Figure 3(a) shows the number of accurate schemas that each approach achieved. It appears that there is no outstanding strategy, i.e., there does not exist a strategy that can always perform better or worse than others. For example, for the testing object 1, *ILP* performed the best in obtaining the accurate MFS, while for the testing object 2, *distinguishing failures* identified the most accurate MFS and for testing object 3, *regarded as one failure* did the best.



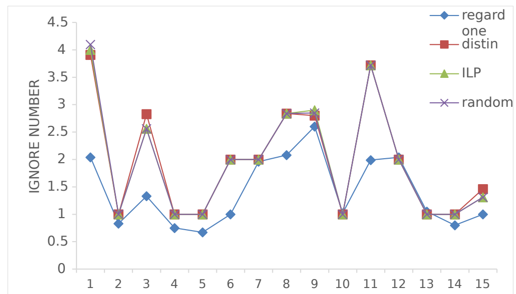
(a) Result of the number of accurately identified MFS



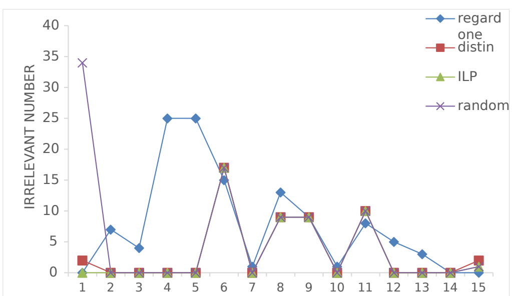
(b) Result of the number of identified sub-schemas of the MFS



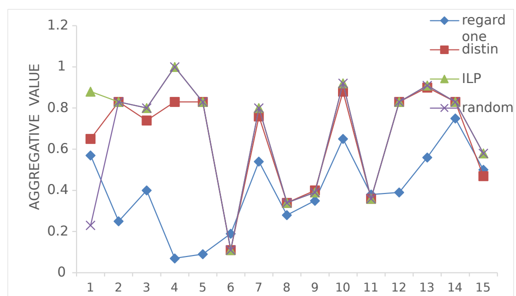
(c) Result of the number of identified super-schemas of the MFS



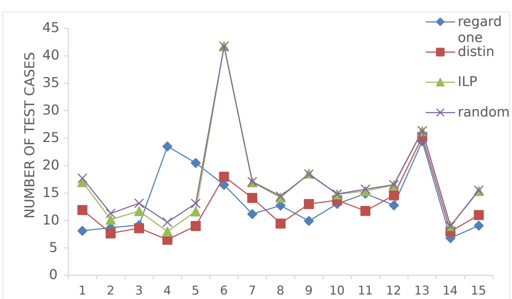
(d) Result of the number of ignored MFS



(e) Result of the number of identified irrelevant schemas of the MFS



(f) The aggregative result for the five metrics



(g) Needed test cases

Fig. 3. Result of the evaluation for second case study

We can further find that strategies *replacement strategy based on ILP searching* (short for *ILP* later) and *distinguishing failures* have similar results. This can be easily understood, as strategy *ILP* is actually a refinement version of the strategy *distinguishing failures*, which also make the failures distinguished from each other. The main difference between *ILP* and *distinguishing failures* 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. As a result, the comparison of other metrics (sub, super, ignore, irrelevant numbers) also showed the similarity between strategy *ILP* and *distinguishing failures*.

Sub number & super number: Figure 3(b) and 3(c) depicts the results for *sub number* and *super number*, respectively. These two figures firstly showed a clear trend for strategies *regarded as one 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.

Although offering similar result as *distinguishing failures* strategy, our strategy *ILP* tend to identify fewer sub schemas and super schemas of actual MFS than strategy *distinguishing failures* (testing objects 1, 9 in Figure 3(b) and testing objects 3, 4,7,9,10 13 in Figure 3(c)). We believe this is an improvement, as too many sub schemas and super schemas will make it harder to identify the actual MFS. One issue is the redundancy problem, as many sub or super schemas in fact point to the same actual MFS.

Ignore number & irrelevant number: The results of the two negative performance metrics are given in Figure 3(d) and 3(e), respectively. One observation is that, comparing with strategy *regarded as one failure*, *distinguishing failures* obtained fewer irrelevant schemas, but ignored more actual MFS. This is also consistent with formal analysis in Section 4.

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* at testing object 3 and 15 in Figure 3(d), but is not as good as strategy *regarded as one failure*. In fact, strategy *regarded as one 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*, we can find that our approach is the best among all three strategies (better than *distinguishing failures* at testing object 1 in Figure 3(e), and better than strategy *regarded as one failure* for most 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 3(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 one failure* (See the testing object 1, 3 and 4 in Figure 3(f)).

It is as expected that *ILP* performed better than *distinguishing failures* as it is actually the refinement version of latter. It is a bit of surprise to find, however, that strategy *distinguishing failures* performed better than *regarded as one failure* at almost all the testing objects. This result cannot be derived from the formal analysis. A possible explanation is that in the constructed testing objects, the possibility of triggering a masking effect is relatively small. Consequently if we take the strategy *regarded as one failure*, we are more likely to misjudge a test case which triggered other failures to be the failing test case for the failure we currently focus on.

Test cases: The number of test cases generated for identifying the MFS indicates the cost of FII approach. The result is shown in Figure 3(g). We can find that strategy *ILP* generated more test cases than the other strategies. In specific, the gap between the *ILP* and other two strategies ranged from about 2 to 5 (except for the 6th testing object, which exceeds 20). 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 one failure*, there is no significant difference between the number of test cases generated.

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

1) *Distinguishing failures* strategy obtained more *super number* and *ignored number* than *regard one 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 need more test cases than the other two strategies, it is acceptable.

6.3. Evaluating the ILP-based test case searching method

The third empirical study aims to evaluate the efficiency of the ILP-based test case searching component of our approach. To conduct this study, we implemented an FII approach which is also augmented by the *replacing test cases* strategy, with test case randomly replaced.

6.3.1. Study setup. The setup of this case study is based on the second case study, and uses the same SUT model as shown in Table XIX. We apply the new random searching based FII approach to identify the MFS in the prepared SUTs. To avoid the bias coming from the randomness, we repeat the new approach 30 times to identify the MFS in each failing test case. We then compute the average additional test cases as well as other metrics listed in section 6.2.1 for the random-based approach.

6.3.2. Results and discussion. The evaluation of this random-based approach is also shown in Figure 3, in which the polygonal line marked with '$\times$' in each sub-figure indicates the results. The raw data can also be found in the column 'R' of Table XX in the appendix.

Compared to the ILP-based approach, we can firstly observe that there is little distinction between them in terms of the metrics: accurate schemas, super-schemas, sub-schemas, ignored schemas, irrelevant schemas (for some particular cases the ILP-based approach performs slightly better, e.g., in Figure 3(b) for the first testing object, the ILP-based approach identified fewer sub schemas than that of the Random-based approach and in Figure 3(c) still for the first object the ILP-based approach identified fewer super schemas than that of the random-based approach). The similar quality of the identified MFS between these two approaches is conceivable as they both use the *test case replacement* strategy, although the test cases generated may be different.

Secondly, when considering the cost, we find that the ILP-based approach performs better, which can reduce on average 1 to 2 test cases compared to the random-based procedure. To be precise, we next compared the test cases only used for replacement of the two approaches. By this we can eliminate the interference of other test cases for identifying MFS, and focus on the performance of the key part of these two approaches — replacement strategy.

Figure 4 shows the result. The data is normalized so that the results of the 15 objects can be shown together. For all subjects ILP reduce the test cases for replacement by about 5 to 40 percent when compared with the random-based approach. On average, for each call of the replacement algorithm, it can save about 0.44 test cases (labeled in each dot of the line). This may be trivial. But considering the masking status in Table XVIII, this cost reduction is significant for MFS identifying in practice.

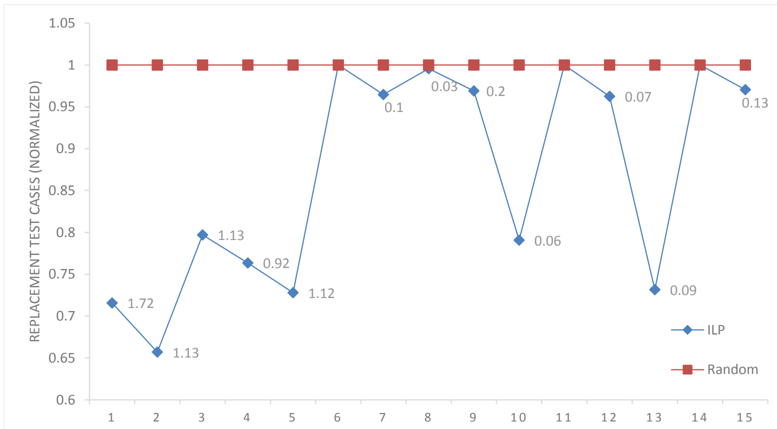


Fig. 4. The comparison of the number of replacement test cases

In summary, the answer for **Q3** is that searching for a satisfied test case affects the performance of our approach, especially regarding the number of extra test cases, and the ILP-based test cases can handle the masking effects at a relatively smaller cost than the random-based approach.

6.4. Comparison with Feedback driven combinatorial testing

The *FDA-CIT* [Yilmaz et al. 2014] approach can handle masking effects so that the generated covering array 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 our comparative approach, as Yilmaz claims that it performs better than the former [Yilmaz et al. 2014].

The FDA-CIT process starts with generating a t-way 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 (Wekas implementation of C4.5 algorithm(J48) [Hall et al. 2009]) 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 these MFS and then repeat the previous steps.

The main target of FDA-CIT is to produce the generated test cases that 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.

6.4.1. 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 covering arrays and fed them to the FDA-CIT. Then 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 generated covering array. Specifically, for each failing test case 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 that are identified by our approach, and evaluate them according to the corresponding metrics. In addition, we recorded the overall test cases (including the initially generated covering array) that this approach needed and compared the magnitude of these test cases with that of FDA-CIT.

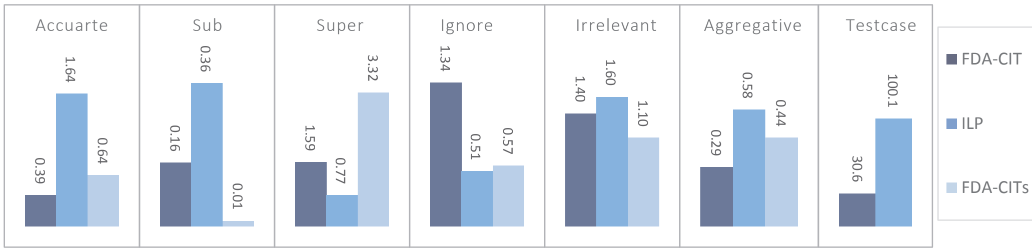
As mentioned before, the FII approach in FDA-CIT, i.e., classification tree algorithm, is a post-analysis technique. Given different test sets, the results identified by the classification tree algorithm are also different. Then a natural question is, what the schemas identified by FDA-CIT will be if the classification tree method is applied on the test cases generated by our ILP approach? This question is of importance as first, we can learn whether the test cases generated by ILP can help FDA-CIT approach to improve the quality of the identified schemas; second, the comparison between ILP and FDA-CIT will be more fair as they share the same test cases. For this, a new approach that is based on FDA-CIT is introduced, which is augmented by replacing the original test cases in FDA-CIT with those generated by ILP approach. Then the schemas identified by the classification tree algorithm in FDA-CIT are recorded and evaluated. This new approach is referred to as *FDA-CITs* later.

6.4.2. Result and discussion. The result is shown in Figure 5. We conducted three groups of experiments. The first one generated 30 different 2-way cover arrays for each testing object, and then for each covering array we applied the three approaches to identify the MFS. The average evaluation results for the experiments based on 30 covering arrays are listed in the Sub-figure 5(a). The other two groups of experiments starts with 3-way covering arrays and 4-way covering arrays, of which their results are depicted in Sub-figure 5(b) and 5(c) respectively.

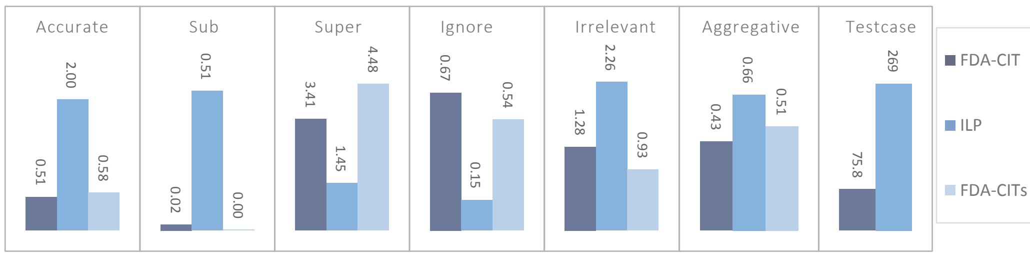
In each sub-figure, there are 7 columns, showing the outcomes for the previous mentioned 6 metrics and one more metric (Column *Testcase*), which indicates the overall test cases that each approach needed. Each column has three bars (Except for the Column *Testcase*, as the overall test cases for ILP and FDA-CITs are the same), which indicate the results for approach FDA-CIT, ILP and FDA-CITs, respectively.

Note that in Figure 5, the results for each metric is the average evaluation for all the results of the experiments on the 15 testing objects in Table XIX. The raw results for each testing object are listed in Table XXI in the appendix . The raw data is organised the same way as Table XX, except that we added a column t which indicates the strength of the covering array generated for this experiment.

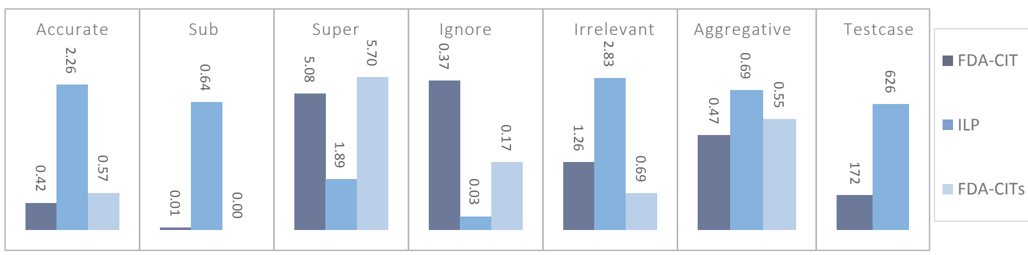
With respect to the relationships between the results and the degree t of the covering arrays, we have the following observations:



(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. 5. Three approaches augmented with the replacing strategy

First, for every metric in our study, the order of the performance of each approach is stable against the change of degree t. Take for example the metric *accurate number*. No matter what t is (2, 3 or 4), *ILP* always obtained the most schemas that are identical to the actual MFS, then is *FDA-CITs*, and the last is *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.

Second, with increasing t, the overall performance of each approach is improved. For example, the score of the aggregative metric of *ILP* is 0.55, 0.66 and 0.69, respectively, for t equals to 2, 3 and 4. The improvement is mainly because with increasing t, the number of test cases also increased. Accordingly, the approach will observe more failing test cases, so that we can get the schemas more close to the actual MFS.

Third, for different approaches in our study, the effect of the change of t on the scores of other metrics varies. Specifically, for *ILP*, with increasing t, metrics *accurate number, sub number, super number, irrelevant number* also increase, while metric *ignore number* decreases. This is mainly because *ILP* is based on *FIC_BS* [Zhang and Zhang 2011], which works on single failing test case. When t increases, the number of test cases also increases. Then when applying our approach, more schemas may be identified from those additional failing test cases, so the number of accurate MFS, sub-schemas of the actual MFS, super-schemas of the actual MFS, and schemas that are irrelevant to the actual MFS will increase. Furthermore, some actual MFS that had been ignored before may be obtained. For *FDA-CIT* and *FDA-CITs*, however, we find that *sub number* and *irrelevant number* decrease with increasing t. We believe this result is due to the use of the classification tree method. A typical classification tree works by partitioning the test cases according to some aspects. Here, the aspect is the parameter value of the SUT. And one path (conjunction of nodes from the root to one leaf in the tree) in this tree is deemed as an MFS. So when test cases increase, the classification may need more nodes to classify the test cases. This induces the so-called 'over fitting' problem. As a result, the schemas identified by *FDA-CIT* and *FDA-CITs* tend to be the super schemas of the actual MFS, leading to a decrease of *sub number* and *irrelevant number*.

Other observations include:

First, compared with the original approach *FDA-CIT*, *FDA-CITs* has obvious advantages at almost all the metrics except for *super number*. In detail, *FDA-CITs* obtained more schemas that are identical to the actual MFS (*accurate number*), fewer schemas that are the sub schemas of actual MFS (*sub number*), and lower scores for the two negative metrics (*ignored number* and *irrelevant number*). At last, the schemas identified by *FDA-CITs* showed an overall higher quality than that of the original *FDA-CIT* (*aggregative metric*). We have discussed previously that *FDA-CIT* tends to identify super-schemas of actual MFS when the test cases increase. So for metric *super number*, it is no surprise that *FDA-CITs* identified more super schemas of actual MFS than *FDA-CIT*, because it used the test cases generated by *ILP*, which were more than that of the original FDA-CIT. The difference between the overall performance of *FDA-CIT* and *FDA-CITs* is also expected. In fact, this result is consistent with our previous observation that when t increases, the overall performance for each approach also increases.

Second, in terms of the quality of the MFS identified, we can clearly find that our approach performed better than the other approaches. This is manifested in that our approach obtained more accurate schemas and identified fewer irrelevant ones. We believe this gap is mainly caused by the FII approach. Because for *ILP* and *FDA-CITs*, the test cases used to identify the MFS are the same. The only difference is how they utilize them to identify MFS. However, this result does not mean that FIC_BS is better than the classification tree method under all conditions. The classification tree method has its own advantage, i.e., it does not need to generate additional test cases, and as a result, *FDA-CIT* generated fewer test cases than that of *ILP*.

In fact, another reason that our approach generated more test cases is that the FII approach, i.e., FIC_BS, works on single test case, so when there are many failing test cases in the covering array, we need to repeatedly use our approach to identify the MFS for each failing test case. This process may produce many redundant test cases, because many failing test cases contain the same MFS, and when we have already identified the MFS in one test case, there is no need to identify it again in other failing test cases. Jieli [Li et al. 2012] introduced a method that utilizes the previous generated test cases to reduce such redundance. Here we did not use this technique to simplify our experiment. We believe if we utilize the MFS that has been already identified in previous iteration, the overall number of test cases will decrease.

Above all, we can conclude three points in this experiment, which provide answer to **Q4**:

1)The degree t of the covering array does not affect the order of the performance of different approaches, but for each approach, the bigger the t is, the better its performance.

2)When taking the test cases generated by our *ILP* approach, *FDA-CITs* performed better than the original *FDA-CIT* approach.

3)Considering the quality of the MFS each approach identified, *ILP* performed better than the other two approaches, although it needed more test cases.

Based on these observations, a recommendation for selecting masking handling techniques in practice is that to get a more precise identification of the MFS in the SUT, *ILP* is preferred, and for lower number of test cases, *FDA-CIT* may be a better choice.

6.5. Threats to validity

6.5.1. 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 make the observed failing test cases and inferred failing test cases different. 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 15 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 injected additional types of faults 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 different FII approaches when applying our test case replacing strategy, we need to try more FII approaches in the future.

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

Another important threat is that our approach is based on the assumption 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. 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 one failure* strategy. With this strategy, we must aware that the MFS identified are likely to be sub-schemas or irrelevant schemas of the actual MFS.

The third threat comes from the possible masking relationships between multiple failures in the real software. In this paper, we just focus on the condition that the masking effects are transitive, i.e., if failure A masks B, and failure B masks C, then failure A must mask failure C. In practice, the relationships between multiple failures may be more complicated. One possible scenario is that two failures 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.

7. RELATED WORKS

Shi and Nie presented an approach for failure revealing and failure diagnosis in CT [Shi et al. 2005], 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 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 [Wang et al. 2010].

Nie et al. 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 SUT [Nie and Leung 2011a]. This approach mutates one value for that parameter, hence generating a group of additional test cases each time to be executed. Compared with SOFOT, this approach strengthens the validation of the factor under analysis and 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. also proposed a similar approach that can efficiently identify the failure-inducing interactions that has no overlapped part [Zhang and Zhang 2011]. Later, Li improved the delta-debugging based approach by exploiting useful information in the executed covering array [Li et al. 2012].

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

Ghandehari et al. defined the suspiciousness of tuple and suspiciousness of the environment of a tuple [Ghandehari et al. 2012]. Based on this, they ranked the possible tuples and generated the test configurations. They further utilized the test cases generated from the inducing interaction to locate the fault [Ghandehari et al. 2013].

Yilmaz proposed a machine learning method to identify inducing interactions from a combinatorial testing set [Yilmaz et al. 2006]. 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 become 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 constrains, higher-strength covering arrays with early failure detection are practical.

Besides, there are additional works that aim to study the impacts 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 with considering the constraints. [Yilmaz 2013] introduced the test case-specific constraints; differing from the system-wide constraints, this constraint can only be triggered in some specific test cases.

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

Dumlu and Yilmaz proposed a feedback-driven approach to work around the masking effects [Dumlu et al. 2011]. 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 further extended their work [Yilmaz et al. 2014] by proposing a multiple-class CTA approach to distinguish failures in SUT. In addition, they empirically studied the impacts of masking effects on both ternary-class and multiple-class CTA approaches.

These works can be categorized into 3 groups according to their relationships with our work. First, the works that aim to identifying the MFS in the SUT. Our work also focuses on identifying the MFS, but instead of single failure, our work considers the impacts of multiple failures 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 that is most similar to our work [Yilmaz et al. 2014], 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.

8. CONCLUSIONS

Masking effects of multiple failures in SUT can bias the results of traditional failure-inducing interactions identifying 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 one fault* 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. The comparison between our approach and traditional approaches was performed on several open-source software. The results indicated our strategy assists the traditional FII approach in achieving better performance when facing masking effects in 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, though our approach required more test cases than FDA-CIT.

As for the future work, we need 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 failures, 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 failures.

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.

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.

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.

Kiran Shakya, Tao Xie, Nuo Li, Yu Lei, Raghu Kacker, and Richard Kuhn. 2012. Isolating Failure-Inducing Combinations in Combinatorial Testing using Test Augmentation and Classification. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 620–623.

Liang Shi, Changhai Nie, and Baowen Xu. 2005. A software debugging method based on pairwise testing. In *Computational Science–ICCS 2005*. Springer, 1088–1091.

Charles Song, Adam Porter, and Jeffrey S Foster. 2012. iTree: Efficiently discovering high-coverage configurations using interaction trees. In *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 903–913.

Ziyuan Wang, Baowen Xu, Lin Chen, and Lei Xu. 2010. Adaptive interaction fault location based on combinatorial testing. In *Quality Software (QSIC), 2010 10th International Conference on*. IEEE, 495–502.

Cemal Yilmaz. 2013. Test case-aware combinatorial interaction testing. *Software Engineering, IEEE Transactions on* 39, 5 (2013), 684–706.

Cemal Yilmaz, Myra B Cohen, and Adam A Porter. 2006. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on* 32, 1 (2006), 20–34.

Cemal Yilmaz, Emine Dumlu, 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.

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 failures

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

A. THE DETAIL OF THE EXPERIMENTS

Table XX. Result of the evaluation

Subject	accurate				sub				super				ignore				irrelevant				overall				test cases			
	O [1]	D [2]	I [3]	R [4]	O	D	I	R	O	D	I	R	O	D	I	R	O	D	I	R	O	D	I	R	O	D	I	R
HSQLDB 2cr8	2	3	5	5	3	2	0	2	0	2	4	5	0(2.04)	0(3.91)	0(4.0)	0(4.1)	0	2	0	34	0.57	0.65	0.88	0.23	8.125	11.92	17	17.72
HSQLDB 2.2.5	2	2	2	2	1	0	0	0	0	1	1	1	0(0.83)	0(1.0)	0(1.0)	0(1.0)	7	0	0	0	0.25	0.83	0.83	0.83	8.67	7.67	10.17	11.3
HSQLDB 2.2.9	2	3	2	2	3	1	1	1	0	4	1	1	0(1.33)	0(2.83)	0(2.56)	0(2.56)	4	0	0	0	0.4	0.74	0.8	0.8	9.167	8.61	11.72	13.14
JFlex 1.4.1	2	2	2	2	0	0	0	0	0	1	0	0	0(0.75)	0(1.0)	0(1.0)	0(1.0)	25	0	0	0	0.07	0.83	1	1	23.5	6.5	8	9.68
JFlex 1.4.2	2	2	2	2	1	0	0	0	0	1	1	1	0(0.67)	0(1.0)	0(1.0)	0(1.0)	25	0	0	0	0.09	0.83	0.83	0.83	20.5	9	11.67	13.12
synthez 1	2	1	1	1	2	0	0	0	0	2	2	2	0(1.0)	0(2.0)	0(2.0)	0(2.0)	15	17	17	17	0.19	0.11	0.11	0.11	16.5	18	41.75	41.75
synthez 2	3	3	3	3	10	0	0	0	0	10	6	6	0(1.96)	0(2.0)	0(2.0)	0(2.0)	1	0	0	0	0.54	0.76	0.8	0.8	11.19	14.12	16.96	17.08
synthez 3	4	3	3	3	4	2	2	2	0	5	5	5	0(2.08)	0(2.84)	0(2.84)	0(2.84)	13	9	9	9	0.28	0.34	0.34	0.34	12.73	9.46	14.18	14.44
synthez 4	3	3	3	3	10	3	2	3	0	6	5	5	0(2.6)	0(2.8)	0(2.9)	0(2.85)	9	9	9	9	0.35	0.4	0.39	0.39	9.91	13.02	18.55	18.45
synthez 5	2	2	2	2	4	0	0	0	0	2	1	1	0(1.02)	0(1.0)	0(1.0)	0(1.0)	1	0	0	0	0.65	0.88	0.92	0.92	13.04	13.7	14.77	14.84
synthez 6	2	3	3	3	15	4	4	4	0	8	8	8	0(1.99)	0(3.72)	0(3.72)	0(3.72)	8	10	10	10	0.38	0.36	0.36	0.36	14.91	11.75	15.37	15.71
synthez 7	3	3	3	3	10	0	0	0	0	6	6	6	0(2.04)	0(2.0)	0(2.0)	0(2.0)	5	0	0	0	0.39	0.83	0.83	0.83	12.77	14.59	16.44	16.53
synthez 8	2	2	2	2	4	0	0	9	0	4	3	3	0(1.05)	0(1.0)	0(1.0)	0(1.0)	3	0	0	0	0.56	0.9	0.91	0.91	24.45	25.25	26.27	26.37
synthez 9	2	1	1	1	1	0	0	0	0	1	1	1	0(0.8)	0(1.0)	0(1.0)	0(1.0)	0	0	0	0	0.75	0.83	0.83	0.83	6.8	8	9	9
synthez 10	0	1	1	1	3	1	1	1	0	0	0	0	0(1.0)	0(1.46)	0(1.31)	0(1.31)	0	2	1	1	0.5	0.47	0.58	0.58	9.08	11	15.38	15.53

[1] O denotes the strategy *regarded as one failure*.
[2] D denotes the strategy *distinguishing failures*.
[3] I denotes the replacement strategy based on ILP searching.
[4] R denotes the replacement strategy based on randomly searching.

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

Table XXI. Comparison with FDA-CIT

Subject		**accurate**			**sub**			**super**			**ignore**			**irrelevant**			**overall**			**test cases**		
	t	F [1]	I [2]	Fs [3]	F	I	Fs	F	I	F	F	I	Fs	F	I	Fs	F	I	Fs	F	I	Fs
HSQL2cr8	2	0.17	2.27	1.57	0.57	0	0	0.17	0.4	2.17	3.87	2.3	2	2.53	0	1.97	0.12	0.51	0.39	23.6	70.1	70.1
	3	1.47	3.67	1	0	0	0	4.67	2	6.07	0.63	0.3	0.17	3	0	1.47	0.51	0.87	0.6	76.6	241.8	241.8
	4	0.83	4.8	1	0	0	0	9.03	3.37	8	0	0	0	0.97	0	0	0.65	0.9	0.71	183.5	606.6	606.6
HSQL2.2.5	2	1	1.97	0.37	0	0	0	2.4	0.73	3.8	0.4	0	0	1.4	0	0.1	0.38	0.87	0.56	26.7	68.8	68.8
	3	0	2	0.4	0	0	0	5	1	3.8	0	0	0	0	0	0	0.52	0.83	0.56	67	202.4	202.4
	4	0	2	0.33	0	0	0	5	1	4	0	0	0	0	0	0	0.53	0.83	0.56	130.1	503.3	503.3
HSQL2.2.9	2	0.9	1.77	0.9	0	0.77	0	1.47	0.47	6.8	1.93	0.53	0	2.37	0	0.2	0.28	0.72	0.58	29.2	78.3	78.3
	3	1	2	0.83	0	1	0	5.13	0.93	7.1	0.2	0	0	0.1	0	0	0.61	0.8	0.61	72.8	221.7	221.7
	4	1	2	1	0	1	0	5.87	1	6.7	0	0	0	0	0	0	0.64	0.8	0.62	129.8	560.3	560.3
JFlex 1.4.1	2	0	2	0	0	0	0	4.03	0	4	0	0	0	0	0	0	0.49	1	0.5	30.5	87.3	87.3
	3	0	2	0	0	0	0	4	0	0	0	0	4	0	0	0	0.5	1	0.5	73.4	269.2	269.2
	4	0	2	0	0	0	0	4	0	0	0	0	0	0	0	0	0.5	1	0.5	190.6	724.7	724.7
JFlex 1.4.2	2	0.3	1.97	0.93	0	0	0	3.6	1	2.16	0.03	0	0	0.63	0	0	0.5	0.83	0.62	34.3	106.9	106.9
	3	0	2	0.97	0	0	0	5	1	2.1	0	0	0	0.03	0	0	0.52	0.83	0.61	72.3	305.7	305.7
	4	0	2	1	0	0	0	5	1	2	0	0	0	0	0	0	0.53	0.83	0.61	186.8	836.9	836.9
synthez 1	2	0.97	1	1	0	0	0	1.7	1.93	2	0	0.07	0	0.33	14.3	0	0.66	0.13	0.78	40.3	342.87	342.87
	3	1	1	1	0	0	0	2	2	2	0	0	0	0	16.73	0	0.78	0.12	0.78	93.4	809.1	809.1
	4	1	1	1	0	0	0	2	2	2	0	0	0	0	17	0	0.78	0.12	0.78	218.8	1532.8	1532.8
synthez 2	2	0.17	1.3	0.73	0.37	0	0	0	0.4	2.37	2.27	1.2	1.03	1.37	0	1.2	0.11	0.52	0.4	19.77	54.4	54.4
	3	0.73	2.23	0.5	0	0	0	1.9	1.3	7.1	1.2	0.43	0.53	2.2	0	1.33	0.36	0.82	0.52	59.5	171.5	171.5
	4	0.63	2.97	0.1	0	0	0	5.3	2.33	16.1	0.53	0	0	2.6	0	1	0.44	0.89	0.54	152.7	415.1	415.1
synthez 3	2	0.43	2.97	0.73	0	0.93	0	4.3	1.73	5.3	0.47	0.17	0.5	1.03	3.77	1.13	0.37	0.46	0.37	48.6	138.7	138.7
	3	0.2	3	0.87	0	1.57	0	7.2	3.67	6.57	0.07	0	0	0.83	6.77	0.07	0.38	0.38	0.44	106.3	315.3	315.3
	4	0.03	3	1	0	1.97	0	10.4	3	6	0	0	0	0.43	8.56	0	0.38	0.34	0.45	147.9	565.7	565.7
synthez 4	2	0.3	2.3	0.33	0.07	0.63	0	2.63	1.97	7.7	1.93	0.63	0.4	3.4	1.4	1.97	0.24	0.6	0.44	42.7	142.2	142.2
	3	0.37	2.97	0.07	0	1.26	0	6.5	3.53	10.97	0.83	0.07	0	2.5	3.43	1.03	0.39	0.54	0.51	86.5	373.2	373.2
	4	0.07	3	0	0	1.77	0	11.7	4.67	11.4	0	0	0	1.33	6.73	0.03	0.48	0.44	0.55	202.2	899.7	899.7
synthez 5	2	0.2	1.2	0.8	0.3	0	0	0.1	0.03	0.83	1.4	0.77	0.97	0.7	0	1	0.2	0.59	0.4	21.9	46.9	46.9
	3	0.87	1.4	0.53	0	0	0	0.5	0.23	3.03	1	0.6	0.77	0.37	0	1.63	0.46	0.71	0.43	76.9	150.3	150.3
	4	0.7	1.9	0.37	0	0	0	1.77	0.33	6.5	0.9	0.1	0.03	1.87	0	2.03	0.34	0.92	0.54	232.9	433.2	433.2
synthez 6	2	0.23	2.63	0.17	0.2	2	0	2.93	1.63	9.63	2.6	0.5	0.4	3.03	3.7	2	0.19	0.42	0.37	45.7	132.6	132.6
	3	0.1	3	0.1	0	2.83	0	7.4	3.83	12.5	1.2	0.17	0.03	2.3	6.5	0.67	0.31	0.38	0.43	99.5	338.9	338.9
	4	0	3	0	0	3.8	0	10.2	6.03	14.5	0.47	0	0	1.8	9.1	0.03	0.37	0.36	0.44	152.6	781.9	781.9
synthez 7	2	0.13	1.43	0.83	0.23	0	0	0.1	0.63	1.4	2.53	1.03	0.93	1.93	0	1.97	0.09	0.61	0.38	20.3	58.8	58.8
	3	0.87	2.17	0.93	0	0	0	0.43	1.23	2.97	1.77	0.17	0.13	3.2	0	2.87	0.2	0.88	0.44	52.6	164.7	164.7
	4	1	2.87	1	0	0	0	3.23	2.53	4.6	0.27	0	0	4.5	0	2.27	0.35	0.9	0.51	145.3	413.1	413.1
synthez 8	2	0	0.2	0.17	0.03	0	0	0	0	0.03	0.3	0.13	0.13	0.17	0	0.3	0.01	0.1	0.05	16.1	45.2	45.2
	3	0	0.6	0.5	0.1	0	0	0	0	0.03	0.97	0.47	0.53	0.63	0	0.87	0.02	0.3	0.17	43.1	64.3	64.3
	4	0	1.33	0.8	0.1	0	0	0	0.07	0.67	1.53	0.4	0.5	1.4	0	0.93	0.04	0.67	0.41	109.3	145.6	145.6
synthez 9	2	1	1	1	0	0	0	0.46	0.6	0.77	0.53	0	0.23	0.63	0.6	0.67	0.54	0.7	0.6	36.2	43.4	43.4
	3	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0.83	0.83	0.83	84.3	145	145
	4	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0.83	0.83	0.83	188	291.6	291.6
synthez10	2	0	0.63	0	0.6	1	0.2	0	0	0.83	1.8	0.37	1.9	1.5	0.3	3.97	0.23	0.61	0.17	23.4	84.9	84.9
	3	0.07	0.97	0	0.23	1	0.03	0.36	0	1.9	2.23	0.03	1.97	4.03	0.53	3.97	0.13	0.66	0.2	73.4	263.2	263.2
	4	0	1	0	0.07	1	0	1.7	0	2	1.87	0	2	4.03	1	4	0.21	0.58	0.2	202.2	685.9	685.9

[1] F denotes the FDA-CIT approach.
[2] I denotes for the our approach with replacement strategy based on ILP searching.
[3] Fs denotes for the FDA-CITs approach.