CMS: An efficient and effective MFS Identification method *

Xintao Niu
State Key Laboratory for Novel Software Technology
Nanjing University
China, 210023
niuxintao@gmail.com

Changhai Nie
State Key Laboratory for Novel Software Technology
Nanjing University
China, 210023
changhainie@nju.edu.cn

Hareton Leung
Department of computing
Hong Kong Polytechnic University
Kowloon, Hong Kong
hareton.leung@polyu.edu.hk

Jeff Lei
Department of Computer Science and Engineering
The University of Texas at Arlington
ylei@cse.uta.edu

Xiaoyin Wang
Department of Computer Science
The University of Texas at San Antonio
Xiaoyin.Wang@utsa.edu

ABSTRACT

Combinatorial testing (CT) aims to detect the failures which are triggered by the interactions of various factors that can influence the behaviour of the system, such as input parameters, and configuration options. Many studies in CT focus on designing an elaborate test suite (called covering array) to reveal such failures. Although covering array can assist testers to systemically check each possible factor interaction, however, it provides weak support to locate the failure-inducing interactions. Recently some elementary researches are proposed to handle the failure-inducing interaction identification problem, but some issues, such as unable to identify overlapping failure-inducing interactions, and generating too many additional test cases, negatively influence the applicability of these approaches. In this paper, we propose a novel failure-inducing identification approach which aims to handle those issues. The key of our approach is to search for a proper factor interaction at each iteration to check whether it is failure-inducing or not until all the interactions in a failing test cases are checked. Moreover, we conduct empirical studies on both widely-used real-life highly-configurable software systems and synthetic softwares. Results showed that our approach obtained a higher quality at the failure-inducing interaction identification, while just needed a smaller number of additional test cases.

CCS Concepts

•**Software defect analysis** → **Software testing and debugging;**

Keywords

Software Testing, Combinatorial Testing, Failure-inducing interactions

1. INTRODUCTION

The behavior of modern software are affected by many factors, such as input parameters, configuration options, and specific events. To test such software system is challenging, as in theory we should test all the possible interaction of these factors to ensure the correctness of the System Under Test (SUT)[19]. When the number of factors is large, the interactions that are needed to check increase exponentially. Hence to apply exhaustive testing is not feasible, and even if it is possible, it is resource-inefficient to check all the interactions. Combinatorial testing (CT) is a promising solution to handle the combinatorial explosion problem [8, 9]. Instead of testing all the possible interactions in a system, it focuses on checking those interactions with number of involved factors no more than a prior number. Many studies in CT focus on designing a elaborate test suite (called covering array) to reveal such failures. Although covering array is effective and efficient as a test suite, it provides weak support to distinguish the failure-inducing interactions from all the remaining interactions.

Consider the following example [1], Table 1 presents a pair-wise covering array for testing an MS-Word application in which we want to examine various pair-wise interactions of options for 'Highlight', 'Status Bar', 'Bookmarks' and 'Smart tags'. Assume the third test case failed. We can get five pair-wise 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,

*(Produces the permission block, and copyright information). For use with SIG-ALTERNATE.CLS. Supported by ACM.

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

Table 1: 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

it is difficult to figure out the specific interactions in this suspicious set that caused the failure. In fact, considering that the interactions consist of other number of factors could also be failure-inducing interactions, e.g., (Highlight: Off) and (Highlight: Off, Status Bar: On, Smart tags: Off), the problem becomes more complicated. Generally, to definitely determine the failure-inducing interactions in a failing test case of n factors, we need to check all the $2^n - 1$ interactions in this test case, which is not possible when n is a large number.

To address this problem, prior work [14] 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 [21], adaptively generating additional test cases according to the outcome of the last test case [25], ranking suspicious interactions based on some rules [5], and using graphic-based deduction [11], among others. These approaches can be partitioned into two categories [2] according to how the additional test cases are generated: *adaptive*–additional test cases are chosen based on the outcomes of the executed tests [18, 14, 5, 16, 25, 17, 20, 10]or *nonadaptive*–additional test cases are chosen independently and can be executed in parallel [21, 2, 11, 12, 24].

These approaches, however, are essentially approximate solutions to failure-inducing interactions identification (Theoretically, a definite solution is of exponential computational complexity). Hence, many issues may affect their effectiveness when applied in practice. Generally, *non-adaptive* approaches can usually accurately identify the failure-inducing interactions, even when there are multiple ones in the SUT. Their effectiveness are based on some mathematical objects [2, 11, 12]. The shortcomings of these approaches are they are very ad-hoc, that is, they have many limitations, such as the number of failure-inducing interactions must be given as well as the number of the maximal factors involved in a failure-inducing interaction. Moreover, these approaches usually consume many test cases [25]. *Adaptive* approaches are much more flexible, and they mainly focus on one failing test case. Commonly they also consume much less test cases than *non-adaptive* approaches. However, some problems, such as unable to handle multiple failure-inducing interactions (especially they have overlapping factors), and cannot handle the newly introduced failure-interactions in the additional generated test cases, negatively affect their performance (in terms of both precision and recall).

In this paper, we propose a novel adaptive failure-inducing interaction identification approach which aims to alleviate these issues. Our approach is based on the notions of *faulty schemas* and *healthy schemas* [1], among which the former will result in a failure if any test case contains it while the latter will not. Furthermore, two important data structures are maintained in our approach, i.e., CMXS (candidate maximal pending schemas) and CMNS (candidate minimal pending schemas). With these two data structures, some schemas in the failing test case can be determined to be healthy, or faulty. Those schemas that cannot be determined by these two data structures will be selected and checked. When an un-determined schema is checked, we will update these two structures. This process is repeated until all the schemas are determined to be healthy or faulty, and the failure-inducing schemas will be selected from those faulty schemas. By doing so, we will not omit any schema that can be candidate failure-inducing interaction in the failing test case. As a result, our approach can handle the cases such as a failing test case containing overlapping failure-inducing interactions and interactions consisting of any different number of factors.

To evaluate the performance of our approach, we firstly conduct several experiments to compare our approach with all the existing failure-inducing schemas identification approaches. These experiments consider various factors that may influence the performance of these approaches, e.g., the number of factors in the SUT, the number of MFS in the SUT, the degrees of these MFS. As far as we are aware, our work is the first to comprehensively compare all those adaptive MFS identification approaches. Moreover, we conduct the experiment based on 9 industrial software with real-life faults. Our results generally suggest that our approach obtains a better failure-inducing schema identification results when compared with other works, while requiring a small number of additional test cases.

Contributions of this paper:

- We proposed several important propositions to support failure-inducing schemas identification.
- We proposed a novel adaptive approach which can identify the failure-inducing schemas effectively and efficiently.
- Our approach takes into account several issues that may negatively affect the performance of existing approaches of failure-inducing schemas identification.
- We conducted a series of experiments to comprehensively compare all those adaptive approaches.

The remainder of this paper is organized as follows: Section 2 introduces some preliminary definitions and propositions. Section 3 describes our approaches for identifying failure-inducing schemas. Section 5 gives the comparisons in theoretical metrics. Section 6 describes the experiment based on real-life subjects. Section 7 summarizes the related works. Section 8 concludes this paper and discusses the future works.

2. PRELIMINARY

This section presents some definitions and propositions to give a formal model for CT.

Assume that the Software Under Test (SUT) is influenced by a set of parameters P, which contains n parameters, and each parameter $p_i \in P$ can take the values from the finite set V_i (i = 1,2,..n).

[1]schema is identical to interaction in this paper, and these two terms may be used interchangeably

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

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

We consider any abnormally executing test case as a *fault*. It can be a thrown exception, compilation error, assertion failure or constraint violation. When faults are triggered by some test cases, it is desired to figure out the cause of these faults.

Definition 2. For the SUT, the τ-set $\{(p_{x_1}, v_{x_1}), (p_{x_2}, v_{x_2}), ..., (p_{x_t}, v_{x_t})\}$, where $0 \leq x_i \leq n$, $p_{x_i} \in P$, and $v_{x_i} \in V_{x_i}$, is called a τ-degree *schema* ($0 < \tau \leq n$), when a set of τ values assigned to τ distinct parameters.

For example, the interactions (Highlight: Off, Status Bar: On, Smart tags: Off) appearing in Section 1 is a 3-degree schema, where three parameters are assigned to corresponding values. In effect a test case itself is a n-degree *schema*, which can be described as $\{(p_1, v_1), (p_2, v_2), ..., (p_n, v_n)\}$.

Note that the schema is a formal description of the interaction between parameter values we discussed before.

Definition 3. Let c_1 be a l-degree schema, c_2 be an m-degree schema in SUT and $l < m$. If $\forall e \in c_1$, $e \in c_2$, then c_1 is the $sub-schema$ of c_2, and c_2 the $super-schema$ of c_2, which can be denoted as $c_1 \prec c_2$.

For example, the 2-degree schema {(Highlight, Off), (Status Bar, On)} is a sub-schema of the 3-degree schema {(Highlight, Off), (Status Bar, On), (Smart tags, Off)}.

Definition 4. If for any test cases that contain a schema, say c, it will trigger a failure, then we call this schema c the *faulty schema*. Additionally, if none of sub-schema of c is a *faulty schema*, we then call the schema c the *minimal failure-causing schema (MFS)* [14].

Note that MFS is identical to the failure-inducing interaction discussed previously. In this paper, the terms *failure-inducing interactions* and *MFS* are used interchangeably. Figuring the MFS helps to identify the root cause of a failure and thus facilitate the debugging process.

Definition 5. A schema, say, c, is called a *healthy schema* when we find at least one passing test case that contains this schema. In addition, if none of super-schema of c is the *healthy schema*, we then call the schema c the *maximal healthy schema (MHS)*.

These two type of schemas, i.e., MFS and MHS, are the keys to our approach, as they are essentially representations of the healthy schemas and faulty schemas in a test case. As shown later, other schemas can be determined to be healthy or faulty by these two type of schemas. As a result, we just need to record these two types of schemas (normally a small amount) instead of recording all the schemas in a test case (up to 2^n) when identifying MFS.

Definition 6. A schema is called a *pending* schema, if it is not yet determined to be *healthy* schema or *faulty* schema.

The *pending* schema is actually the *un-determined* schema discussed in Section 1. In effect, to identify the MFS in a failing test case is to figure out all the *pending* schemas, and then try to classify each of them into *healthy* or *faulty* schema. After that, the MFS can be selected from those *faulty* schemas by definition.

To facilitate our discussion, we introduce the following assumptions that will be used throughout this paper:

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

This assumption is a common assumption of CT[25, 5, 16]. It indicates that the outcome of executing a test case is reproducible and will not be affected by some random events. Some approaches have already proposed measures to handle this problem, e.g., studies in [21, 3] use multiple covering arrays to avoid this problem.

ASSUMPTION 2. *If a test case contains a MFS, it must fail as expected.*

This assumption shows that we can always observe the failure caused by the MFS. In practice, some issues may prevent this observation. For example, the coincidental correctness problem [13] may happen through testing, when the faulty-code is executed but the failure doesn't propagate to the output. Masking effect [22] may also make the failure-observation difficult, as other failure may triggered and stop the program to go on discovering the remaining failures.

We will later discuss the impacts on MFS identification from these two assumptions, as well as how to alleviate them. Based on these definitions and assumptions, we can get several propositions as following. These propositions are the foundation of our approach, and their proofs are omitted due to their simplicity.

PROPOSITION 1. *Given schemas s_1, s_2, and s_3, if $s_1 \prec s_2$, $s_2 \prec s_3$, then $s_1 \prec s_3$.*

PROPOSITION 2. *Given a faulty schema s_1, then $\forall s_2, s_1 \prec s_2$, s_2 is a faulty schema.*

PROPOSITION 3. *Given a healthy schema s_1, then $\forall s_2, s_2 \prec s_1$, s_2 is a healthy schema.*

PROPOSITION 4. *Given two pending schemas s_1, s_2, and $s_1 \prec s_2$. Then $\forall s_3, s_1 \prec s_3 \prec s_2$, s_3 is a pending schema.*

3. THE APPROACH

In this section, we will describe our approach to identify the failure-inducing schemas in the SUT. To give a better description, we will give several propositions which provide theoretical supports for our approach. The proofs of these propositions are given in the Appendix.

3.1 Obtaining pending schema

To identify the MFS in a failing test case, we need to figure out all the pending schemas and then classify them into faulty or healthy schemas. For this, we firstly need to be able to find one pending schema and check it. Then, we repeat this procedure until no more pending schema can be obtained.

To be general, we formalize the problem of obtaining one pending schema as the following problem:

Given a failing test case t ={(p_1, v_1), (p_2, v_2), ..., (p_n, v_n)}, *a set of faulty schemas* $FSS = \{ c_1, c_2, ...c_i, ... \}$, *where* $c_i \prec t$ *or* $c_i = t$ *(That is,* $c_i \preceq t$*), a set of healthy schemas* $HSS = \{ c_1, c_2, ...c_i, ... \}$, *where* $c_i \preceq t$, *and* $FSS \bigcap HSS = \emptyset$. *The goal is to find one pending schema.*

Note that at the beginning of MFS identification, if there is no additional information, FSS will be initialized to have one element, i.e., t itself, and HSS is an empty set.

We will settle this problem step by step. According to the Propositions 2 and 3, it is easy to find that, the pending schemas set PSS should be the following set

$$PSS = \{c | c \prec t \;\&\&\; \forall c_f \in FSS, c_f \not\preceq c \;\&\&\; \forall c_h \in HSS, c \not\preceq c_h\}. \quad (1)$$

To obtain one pending schema, we just need to select one schema which satisfies $c \in PSS$. However, to directly utilize Formula 1 is not practical to obtain one pending schema, because in the worst case it needs to check every schema in a test case t, of which the complexity is $O(2^n)$. Hence, we need to find another formula which is equivalent to Formula 1, but with much lower complexity.

For this purpose, we defined the following two sets, i.e. CMXS and CMNS.

Definition 7. For a k-degree faulty schema c = {(p_{x_1}, v_{x_1}), (p_{x_2}, v_{x_2}), ..., (p_{x_k}, v_{x_k})}, a failing test case t = {(p_1, v_1), (p_2, v_2), ..., (p_n, v_n)}, and $c \preceq t$. We denote the candidate maximal pending schema as CMXS(c,t)= $\{t \backslash (p_{x_i}, v_{x_i}) \mid (p_{x_i}, v_{x_i}) \in c\}$.

Note that CMXS is the set of schemas that remove one distinct factor value in c, such that all these schemas will not be the super-schema of c. For example assume the failing test case {(p_1, v_1), (p_2, v_2), (p_3, v_3), (p_4, v_4)}, and a faulty schema {(p_1, v_1), (p_3, v_3)}. Then the CMXS set is { {(p_2, v_2), (p_3, v_3), (p_4, v_4)}, {(p_1, v_1), (p_2, v_2), (p_4, v_4)} }. Obviously, the complexity of obtaining CMXS of one faulty schema is $O(\tau)$, where τ is the number of parameter values in this faulty schema, i.e., the degree of this schema.

With respect to the CMXS set of a single faulty schema, we can get the following proposition:

PROPOSITION 5. *Given a faulty schema* c_1, *a failing test case* t, *we have* $\{c \mid c \prec t \;\&\&\; c_1 \not\preceq c\} = \{c \mid \exists c_1' \in CMXS(c_1, t),\ c \preceq c_1'\}$.

In the equation of Proposition 5, the schemas of the left side, i.e., $\{c \mid c \preceq t \;\&\&\; c_1 \not\preceq c\}$, are the sub-schemas of test case t, but not the super-schemas of faulty schema c_1 nor equal to c_1. Note that the pending schemas can only appear in this set, because they cannot be the super-schema of any faulty schema nor equal to them. The right side set in this equation, i.e., $\{c \mid \exists c_1' \in CMXS(c_1, t),\ c \preceq c_1'\}$, are schemas which are sub-schemas of or equal to at least one schema in $CMXS(c_1, t)$. Proposition 5 indicates that these two schema sets are equivalent. As an example, considering a failing test case t – {$(p_1, 1)$,$(p_2, 1)$,$(p_3, 1)$,$(p_4, 1)$}, and a faulty schema c_f – {$(p_3, 1)$}. Table 2 shows the schema set $\{c \mid c \prec t \;\&\&\; c_f \not\preceq c\}$, $CMXS(c_f, t)$ and $c \mid \exists c_1' \in CMXS(c_f, t),\ c \preceq c_1'\}$.

We can extend this conclusion to a set of faulty schemas. For this, we need the following notation: For two faulty schemas c_1, c_2, and a failing test case t ($c_1 \preceq t$, $c_2 \preceq t$), let $CMXS(c_1, t) \bigwedge CMXS(c_2, t) = \{c \mid c = c_1' \cap c_2',$ *where* $c_1' \in CMXS(c_1, t)$, *and* $c_2' \in CMXS(c_2, t)\}$.

For example, let $t = \{(p_1, v_1), (p_2, v_2), (p_3, v_3)\}$, $c_1 = \{(p_1, v_1), (p_2, v_2)\}$, $c_2 = \{(p_2, v_2), (p_3, v_3)\}$. Then we have $CMXS(c_1, t) = \{ \{(p_1, v_1), (p_3, v_3)\}, \{(p_2, v_2), (p_3, v_3)\} \}$, $CMXS(c_2, t) = \{ \{(p_1, v_1), (p_2, v_2)\}, \{(p_1, v_1), (p_3, v_3)\} \}$, and $CMXS(c_1, t) \bigwedge CMXS(c_2, t) = \{\{(p_1, v_1)\}, \{(p_1, v_1), (p_3, v_3)\}, \{(p_2, v_2)\}, \{(p_3, v_3)\}\}$. It is easy to know the complexity of obtaining CMXS of two faulty schemas is $O(\tau^2)$, where τ is the number of parameter values in the faulty schema. Based on this, we denote $CMXS(FSS, t)$ for a set of faulty schemas.

Definition 8. Given a failing test case t = {(p_1, v_1), (p_2, v_2), ..., (p_n, v_n)}, and a set of faulty schemas FSS = $\{c_1, c_2, ...c_i, ...\}$, where $c_i \preceq t$, we denote the candidate maximal pending schema of this set as CMXS(FSS,t)= $\bigwedge_{c_i \in FSS} CMXS(c_i, t)$.

Note to compute the CMXS of a set of faulty schema, we just need to sequentially compute the CMXS of two faulty schemas until the last schema in this set is computed. Hence, the complexity of obtaining CMXS of a set of faulty schema is $O(\tau^{|FSS|})$, where |FSS| is the number of faulty schemas in the schema set, and τ is the degree of the schema. According to Proposition 5, we have:

PROPOSITION 6. *Given a failing test case* $t = \{(p_1, v_1), (p_2, v_2), ..., (p_n, v_n)\}$, *and a set of faulty schemas* $FSS = \{c_1, c_2, ...c_i, ...\}$, *where* $c_i \preceq t$, *we have* $\{c \mid c \preceq t \;\&\&\; \forall c_i \in FSS, c_i \not\preceq c\} = \{c \mid \exists c_1' \in CMXS(FSS, t),\ c \preceq c_1'\}$.

Proposition 6 extends Proposition 5 from a single faulty schema to a set of faulty schemas. As an example, considering a failing test case t – {$(p_1, 1)$,$(p_2, 1)$,$(p_3, 1)$,$(p_4, 1)$}, and a set of faulty schemas FSS – { {$(p_3, 1)$}, {$(p_1, 1)$, $(p_2, 1)$} }. Table 3 shows the schema set $\{c \mid c \prec t \;\&\&\; \forall c_i \in FSS, c_i \not\preceq c\}$, $CMXS(FSS, t)$ and $\{c \mid \exists c_1' \in CMXS(FSS, t),\ c \preceq c_1'\}$.

Next, we give the definition of *CMNS*.

Definition 9. For a k-degree healthy schema c = {(p_{x_1}, v_{x_1}), (p_{x_2}, v_{x_2}), ..., (p_{x_k}, v_{x_k})}, a failing test case t = {(p_1, v_1), (p_2, v_2), ..., (p_n, v_n)}, and $c \prec t$. We denote the candidate minimal pending schema as CMNS(c,t)=$\{(p_{x_i}, v_{x_i}) \mid (p_{x_i}, v_{x_i}) \in t \backslash c\}$.

Note that CMNS is the set of schemas that are assigned to one distinct factor value that is not in c, such that all these schemas will not be the sub-schema of c. For example assume the failing test case {(p_1, v_1), (p_2, v_2), (p_3, v_3), (p_4, v_4)}, and a healthy schema {(p_1, v_1), (p_3, v_3)}. Then the CMNS set is { {(p_2, v_2)}, {(p_4, v_4)} }. With respect to the CMNS set of a single healthy schema, we can get the following proposition:

PROPOSITION 7. *Given a healthy schema* c_1, *a failing test case* t, *we have* $\{c \mid c \prec t \;\&\&\; c \not\preceq c_1\} = \{c \mid c \prec t \;\&\&\; \exists c_1' \in CMNS(c_1, t),\ c_1' \preceq c\}$.

In the equation of Proposition 7, the schemas of the left side, i.e., $\{c \mid c \preceq t \;\&\&\; c \not\preceq c_1\}$, are the sub-schemas of test case t, but not the sub-schemas of healthy schema c_1 nor equal to c_1. It is obvious that the pending schemas can also only appear in this set, because they cannot be the sub-schema of any healthy schema nor equal to them. The right side set in this equation, i.e.,$\{c \mid c \preceq t \;\&\&\; \exists c_1' \in CMNS(c_1, t),\ c_1' \preceq c\}$, are sub-schemas of test case t, and

Table 2: An example of Proposition 5

Test case t	$\{\mathbf{c} \mid \mathbf{c} \prec \mathbf{t}$ && $\mathbf{c_f} \not\preceq \mathbf{c}\}$	**CMXS($\mathbf{c_f}$,t)**	$\{\mathbf{c} \mid \exists \mathbf{c_1'} \in$ **CMXS($\mathbf{c_f}$, t)**, $\mathbf{c} \preceq \mathbf{c_1'}$ $\}$
$\{(p_1, 1),(p_2, 1),(p_3, 1),(p_4, 1)\}$	$\{(p_1, 1),(p_2, 1),(p_4, 1)\}$	$\{(p_1, 1),(p_2, 1),(p_4, 1)\}$	$\{(p_1, 1),p_2, 1),(p_4, 1)\}$
Faulty schema $\mathbf{c_f}$	$\{(p_1, 1),(p_2, 1)\}$		$\{(p_1, 1),(p_2, 1)\}$
$\{(p_3, 1)\}$	$\{(p_1, 1),(p_4, 1)\}$		$\{(p_1, 1),(p_4, 1)\}$
	$\{(p_2, 1),(p_4, 1)\}$		$\{(p_2, 1), (p_4, 1)\}$
	$\{(p_1, 1)\}$		$\{(p_1, 1)\}$
	$\{(p_2, 1)\}$		$\{(p_2, 1)\}$
	$\{(p_4, 1)\}$		$\{(p_4, 1)\}$

Table 3: An example of Proposition 6

Test case t	$\{\mathbf{c} \mid \mathbf{c} \prec \mathbf{t}$ && $\forall \mathbf{c_i} \in FSS, \mathbf{c_i} \not\preceq \mathbf{c}\}$	**CMXS(FSS, t)**	$\{\mathbf{c} \mid \exists \mathbf{c_1'} \in$ **CMXS(FSS, t)** , $\mathbf{c} \preceq \mathbf{c_1'}$ $\}$
$\{(p_1, 1),(p_2, 1),(p_3, 1),(p_4, 1)\}$	$\{(p_1, 1),(p_4, 1)\}$	$\{(p_1, 1),(p_4, 1)\}$	$\{(p_1, 1),(p_4, 1)\}$
Faulty schema set FSS	$\{(p_2, 1),(p_4, 1)\}$	$\{(p_2, 1),(p_4, 1)\}$	$\{(p_2, 1),(p_4, 1)\}$
$\{(p_3, 1)\}$	$\{(p_1, 1)\}$		$\{(p_1, 1)\}$
$\{(p_1, 1),(p_2, 1)\}$	$\{(p_2, 1)\}$		$\{(p_2, 1))\}$
	$\{(p_4, 1)\}$		$\{(p_4, 1)\}$

also are the super-schemas of or equal to at least one schema in $CMXS(c_1, t)$. Proposition 7 indicates that these two schema sets are equivalent. As an example, considering a failing test case t – $\{(p_1, 1),(p_2, 1),(p_3, 1),(p_4, 1)\}$, and a healthy schema c_h – $\{(p_1, 1),(p_2, 1),(p_3, 1)\}$. Table 4 shows the schema set $\{c \mid c \prec t$ && $c_h \not\preceq c\}$, $CMNS(c_h, t)$ and $c \mid c \prec t$ && $\exists c_1' \in CMXS(c_h, t)$, $c_1' \preceq c\}$.

Similarly, for two healthy schemas c_1, c_2, and a failing test case t ($c_1 \preceq t$, $c_2 \preceq t$), let $CMNS(c_1, t) \bigvee CMNS(c_2, t) = \{c \mid c = c_1' \cup c_2'$, *where* $c_1' \in CMXS(c_1, t)$, *and* $c_2' \in CMXS(c_2, t)\}$.

For example, let $t = \{(p_1, v_1), (p_2, v_2), (p_3, v_3)\}$, $c_1 = \{(p_1, v_1), (p_2, v_2)\}$, $c_2 = \{(p_2, v_2), (p_3, v_3)\}$. Then we have $CMNS(c_1, t) = \{ \{(p_3, v_3)\} \}$, $CMNS(c_2, t) = \{ \{(p_1, v_1)\} \}$, and $CMNS(c_1, t) \bigwedge CMNS(c_2, t) = \{\{(p_1, v_1), (p_3, v_3)\}\}$. Based on this, we denote $CMXS(FSS, t)$ for a set of faulty schemas.

Based on this, we denote CMNS(HSS,t) for a set of faulty schemas.

Definition 10. Given a failing test case t = $\{(p_1, v_1)$, (p_2, v_2), ..., $(p_n, v_n)\}$, and a set of healthy schemas HSS = $\{c_1, c_2, ...c_i, ...\}$, where $c_i \preceq t$, we denote the candidate minimal pending schema of this set as CMNS(HSS,t)= $\bigvee_{c_i \in HSS} CMNS(c_i, t)$.

Similar to CMXS(FSS,t), the complexity to obtain CMNS(HSS,t) is $O(\tau^{|HSS|})$, where |HSS| is the number of healthy schemas in the schema set, and τ is the degree of the schema. With respect to CMNS(HSS,t), we have:

PROPOSITION 8. *Given a failing test case* $t = \{(p_1, v_1), (p_2, v_2), ..., (p_n, v_n)\}$, *and a set of healthy schemas* $HSS = \{c_1, c_2, ...c_i, ...\}$, *where* $c_i \preceq t$, *we have* $\{c \mid c \prec t$ && $\forall c_i \in HSS, c \not\preceq c_i\} = \{c \mid c \prec t$ && $\exists c_1' \in CMNS(HSS, t)$, $c_1' \preceq c\}$.

Similar to Proposition 5 and 6, Proposition 8 extends Proposition 7 from a single healthy schema to a set of healthy schemas. As an example, considering a failing test case t – $\{(p_1, 1),(p_2, 1),(p_3, 1),(p_4, 1)\}$, and a set of schema schemas HSS – $\{ \{(p_1, 1),(p_2, 1),(p_3, 1)\}, \{(p_3, 1),(p_4, 1)\} \}$. Table 5 shows the schema set $\{c \mid c \prec t$ && $\forall c_i \in HSS, c \not\preceq c_i\}$, $CMNS(HSS, t)$ and $\{c \mid c \prec t$ && $\exists c_1' \in CMNS(HSS, t)$, $c_1' \preceq c\}$.

Based on Proposition 6 and 8, we can easily learn that $\{c \mid c \prec t$ && $\forall c_i \in FSS, c_i \not\preceq c\} \bigcap \{c \mid c \prec t$ && $\forall c_i \in HSS, c \not\preceq c_i\} = \{c \mid \exists c_1' \in CMXS(FSS, t)$, $c \preceq c_1'\} \bigcap \{c \mid c \prec t$ && $\exists c_1' \in CMNS(HSS, t)$, $c_1' \preceq c\}$. Considering $\forall c \in \{c \mid \exists c_1' \in CMXS(FSS, t)$, $c \preceq c_1'\}$, $c \prec t$, we can transform the aforementioned formula into the following equation.

$\{c \mid c \prec t$ && $\forall c_i \in FSS, c_i \not\preceq c$&& $\forall c_i \in HSS, c \not\preceq c_i\} = \{c \mid \exists c_1' \in CMXS(FSS, t), c \preceq c_1'$ && $\exists c_1' \in CMNS(HSS, t)$, $c_1' \preceq c\} = \{c \mid \exists c_1' \in CMXS(FSS, t), c_2' \in CMXS(FSS, t)$, $c_2' \preceq c \preceq c_1'\}$.

Note that the leftmost side of this equation is identical to Formula 1, hence, we can learn that

$$PSS = \{c \mid \exists c_1' \in CMXS(FSS, t), c_2' \in CMNS(HSS, t), \; c_2' \preceq c \preceq c_1'\}. \tag{2}$$

According to Formula 2, the complexity of obtaining one pending schema is $O(\tau^{|FSS|} \times \tau^{|HSS|})$. This is because to obtain one pending schema, we only need to search the schemas in $CMXS(FSS, t)$ and $CMNS(HSS, t)$, of which the complexity are $O(\tau^{|FSS|})$ and $O(\tau^{|HSS|})$, respectively. Then we need to check each pair of schemas in these two sets, to find whether exists $c_1 \in CMXS(FSS, t)$, $c_2 \in CMNS(HSS, t)$, such that $c_2 \preceq c_1$. If so, then both c_2 and c_1 satisfy Formula 2. Furthermore, $\forall c_3, c_2 \preceq c_3 \preceq c_1$, c_3 also satisfy Formula 2. Hence, the complexity of obtaining one pending schema is $O(\tau^{|FSS|} \times \tau^{|HSS|})$.

In fact, we can further reduce the complexity of obtaining pending schemas. When given a set of schemas S, let the minimal schemas as $S^{\perp} = \{c | c \in S, \nexists c' \in S, c' \prec c\}$ and the maximal schemas as $S^{\top} = \{c | c \in S, \nexists c' \in S, c \prec c'\}$. Based on this, we can have the following proposition:

PROPOSITION 9. *Given a failing test case* t, *a set of faulty schemas* FSS, *and a set of healthy schemas* HSS, *we have* $\{c \mid \exists c_1' \in CMXS(FSS^{\perp}, t)$, $c_2' \in CMNS(\ HSS^{\top}, t)$, $c_2' \preceq c \preceq c_1'\} = \{c \mid \exists c_1' \in CMXS(FSS, t)$, $c_1' \in CMNS(HSS, t)$,

Table 4: An example of Proposition 7

Test case t	$\{c \mid c \prec t \;\&\&\; c_h \not\preceq c\}$	CMNS(c_h, t)	$\{c \mid c \prec t \;\&\&\; \exists c_1' \in CMNS(c_h, t),\ c \preceq c_1'\}$
$\{(p_1, 1),(p_2, 1),(p_3, 1),(p_4, 1)\}$	$\{(p_4, 1)\}$	$\{(p_4, 1)\}$	$\{((p_4, 1)\}$
Heathy schema c_f	$\{(p_1, 1),(p_4, 1)\}$		$\{(p_1, 1),(p_4, 1)\}$
$\{(p_1, 1),(p_2, 1),(p_3, 1)\}$	$\{(p_2, 1),(p_4, 1)\}$		$\{(p_2, 1),(p_4, 1)\}$
	$\{p_3, 1),(p_4, 1)\}$		$\{p_3, 1),(p_4, 1)\}$
	$\{(p_1, 1),(p_2, 1),(p_4, 1)\}$		$\{(p_1, 1),(p_2, 1),(p_4, 1)\}$
	$\{(p_1, 1),(p_3, 1),(p_4, 1)\}$		$\{(p_1, 1),(p_3, 1),(p_4, 1)\}$
	$\{(p_2, 1),(p_3, 1),(p_4, 1)\}$		$\{(p_2, 1),(p_3, 1),(p_4, 1)\}$

Table 5: An example of Proposition 8

Test case t	$\{c \mid c \prec t \;\&\&\; \forall c_i \in HSS, c \not\preceq c_i\}$	CMXS(HSS, t)	$\{c \mid c \prec t \;\&\&\; \exists c_1' \in$ CMNS(HSS, t), $c_1' \preceq c\}$
$\{(p_1, 1),(p_2, 1),(p_3, 1),(p_4, 1)\}$	$\{(p_1, 1),(p_4, 1)\}$	$\{(p_1, 1),(p_4, 1)\}$	$\{(p_1, 1),(p_4, 1)\}$
Heathy schema set HSS	$\{(p_2, 1),(p_4, 1)\}$	$\{(p_2, 1),(p_4, 1)\}$	$\{(p_2, 1),(p_4, 1)\}$
$\{(p_1, 1),(p_2, 1),(p_3, 1)\}$	$\{(p_1, 1),(p_2, 1),(p_4, 1)\}$		$\{(p_1, 1),p_2, 1),(p_4, 1)\}$
$\{(p_3, 1),(p_4, 1)\}$	$\{(p_1, 1),(p_3, 1),(p_4, 1)\}$		$\{(p_1, 1),(p_3, 1),(p_4, 1)\}$
	$\{(p_2, 1),(p_3, 1),(p_4, 1)\}$		$\{(p_2, 1),(p_3, 1),(p_4, 1)\}$

$c_2' \preceq c \preceq c_1'$ }.

According to this proposition, we have the third formula to compute pending schemas as follow:

$$PSS = \{c \mid \exists c_1' \in CMXS(FSS^{\perp}, t), c_2' \in CMNS(HSS^{\top}, t), \; c_2' \preceq c \preceq c_1'\}. \tag{3}$$

According to Formula 3, the complexity of obtaining one pending schema is is $O(\tau^{|FSS^{\perp}|} \times \tau^{|HSS^{\top}|})$, where $|FSS^{\perp}|$ and $|HSS^{\top}|$ is a relatively small number during MFS identification.

3.2 Identify the MFS

To identify the MFS, according to Definition 4, we need to obtain a faulty schema, and make sure that all of its subschemas are healthy schemas. For this aim, we need to obtain the pending schemas and then classify them into faulty or healthy. As we have already given the method to obtain the pending schema, it is easy to get the following algorithm to identify the MFS.

Algorithm 1 Basic MFS identification

Input: t ▷ failing test case
P ▷ parameters of the SUT
Output: MFS ▷ MFS in test case t

```
1: HSS^⊤ ← {}
2: FSS^⊥ ← {t}
3: while true do
4:     Cfs ← CMFS(FSS^⊥, t)
5:     Chs ← CMHS(HSS^⊤, t)
6:     s_p ← obtain(Cfs, Chs)
7:     if s_p == Null then
8:         break
9:     else
10:        check(s_p)
11:        FSS^⊥ ← update(FSS^⊥, s_p)
12:        HSS^⊤ ← update(HSS^⊤, s_p)
13:    end if
14: end while
15: return FSS^⊥
```

In this basic algorithm, $HSS^{\top}$ is the maximal set of healthy schemas, and is initially to be empty (line 1). $FSS^{\perp}$ is the minimal set of faulty schemas, and is initially assigned to have test case t (line 2). In the loop (line 3 - 14), we firstly obtaining one pending schema (line 4 - 6). Note that s_p must satisfy Formula 3, i.e., $s_p \in$ PSS $= \{c \mid \exists c_1' \in Cfs,\ c \preceq c_1' \;\&\&\; \exists c_1' \in Chs,\ c_1' \preceq c\}$. In the implementation of our algorithm, we do not need to obtain all the pending schemas; instead, we just need to get one schema that is in this set. If none schema can be obtained (line 7) (indicating that all the schemas are classified into healthy or faulty), we will break the loop (line 8), and output the MFS (line 15). Otherwise, we will check this schema to be faulty or healthy (line 10), and then update $FSS^{\perp}$ and $HSS^{\top}$ (line 11).

It is noted that, in Algorithm 1, we did not describe which pending schema s_p is obtained from the pending schemas set PSS. How to select one pending schema is important, as it will significantly influence the speed of convergence of this algorithm, as well as number of pending schemas that need to be checked. To better illustrate the complete MFS identification algorithm with pending schema selection, we will first introduce two auxiliary algorithms.

The first algorithm aims at checking all the schemas in a list of ordered pending schemas $\{c_1, c_2, c_i, ..., c_k\}$, where $c_i \prec c_{i+1}$. Note that we do not necessary check them one by one. This is because according to Proposition 2, if one schema is checked to be faulty, then all its super schemas are faulty schemas as well. On the other hand, according to Proposition 3, if one schema is checked to be healthy, then all its subschemas are also healthy schemas.

Combining these two facts together, when given a list of ordered pending schemas $\{c_1, c_2, c_i, ..., c_k\}$, where $c_i \prec c_{i+1}$, the best strategy is to use binary search to check the pending schema in this list. Algorithm 2 shows the detail of such strategy.

The input of Algorithm 2 is a list of ordered pending schemas *list*. The output of this algorithm is index of the minimal faulty schema in this set. In this algorithm, it firstly checks the first schema in *list* (line 1). If this schema is checked to be healthy (line 2), it will return -1 (line 2),

Algorithm 2 Binary-Search-Check

Input: $list$ ▷ List of ordered pending schemas
Output: $fIndex$ ▷ the index of the minimal faulty schema

```
1: if check(list.get(0)) == Healthy then
2:     return −1
3: end if
4: fIndex ← 0
5: head ← 1
6: tail ← list.size() − 1
7: while tail ≥ head do
8:     middle ← (head + tail)/2
9:     s_test ← list.get(middle)
10:    if check(s_test) == Fail then
11:        head = middle + 1
12:        fIndex ← middle
13:    else
14:        tail ← middle − 1
15:    end if
16: end while
17: return fIndex
```

Algorithm 3 Narrow-down-fs

Input: fs ▷ Given faulty schema
t ▷ original failing test case
$HSS^{\top}$ ▷ Current maximal healthy schema set
$FSS^{\perp}$ ▷ Current minimal faulty schema set

```
1: CandiMFS ← fs
2: list ← {}
3: while true do
4:     Cxs ← CMXS(FSS^⊥, t)
5:     Cns ← CMNS(HSS^⊤, t)
6:     list ← getLongest(Cxs, Cns, CandiMFS)
7:     if list == Null then
8:         break
9:     end if
10:    fIndex ← Binary-Search-Check(list)
11:    if fIndex > −1 then
12:        CandiMFS ← list.get(fIndex)
13:        FSS^⊥ ← update(FSS^⊥, CandiMFS)
14:    end if
15:    HSS^⊤ ← update(HSS^⊤, list.get(fIndex + 1))
16: end while
```

which indicates that there are no faulty schemas in this list. Otherwise, it will use binary search strategy to isolate the minimal faulty schema in this set (line 4 - 16). Specifically, it uses head and tail indexes (line 5 - 6) to record the first and last pending schemas, respectively, at each iteration. At each time, it selects the schema with its index in the middle between head and tail indexes (line 8 - 9). If it is checked to be faulty (line 10), we assign the head index to the next location of current middle index (line 11). This is because, all the schemas with index smaller than the middle index are super-schemas of selected schema, which are faulty schemas according to Proposition 2. Additionally, we will assign the index of the minimal faulty schema to the current middle index (line 12). If the schema is checked to be healthy, we will update the tail index to be the previous location of the middle index (line 14). This is because all the schemas after the middle index are subschemas of the selected schema, which are healthy schemas according to Proposition 3.

Note that the minimal faulty schema which are returned by Algorithm 2 is not necessary a MFS. Actually, we only checked all the schemas in the ordered pending schemas. Other sub-schemas of this minimal faulty schema may still be pending schemas. Hence, to obtain the MFS, we need to second auxiliary Algorithm to go on narrowing down the minimal faulty schema. Algorithm 3 depicts the complete process to narrow down a given faulty schema.

The input of this algorithm is a given faulty schema fs, the failing test case t, current maximal healthy faulty schema set $HSS^{\top}$, current minimal faulty schema set $FSS^{\perp}$.

In this algorithm, variable $CandiMFS$ records the currently analyzed minimal faulty schema and is initialed to be fs (line 1). $list$ is a list of subschemas of $CandiMFS$, which is an ordered set of pending schemas as $\{c_1, c_2, c_3, ..., c_m\}$, where $c_i \prec c_{i+1}$. It is initialed to be a empathy set (line 2).

Algorithm 3 loops until the MFS is identified (line 3 - 16). In this loop, it firstly obtain the candidate maximal pending schemas Cxs (line 4), as well as the candidate minimal pending schemas Cns (line 5). Then, an ordered list of subschemas of this candidate MFS is obtained (line 6). More formally, $list = \{c_1, c_2, c_3, ..., c_m\}$, where $c_i \prec c_{i+1}, c_i \in Cns, c_i \in Cxs, c_i \prec CandiMFS$. Note that there may be more than one set of schemas that satisfies our requirement, and each time we only select the set with the maximal number of schemas. The reason why we do this is to fully utilize binary search.

If we cannot obtain any sublist (line 7), which means that all the subschemas of the candidate MFS are checked to be healthy, we will then break the loop (line 8).

Otherwise, we will start Algorithm 2 to check all the schemas in this list (line 10). If the algorithm return a index which is large than -1 (line 11), which indicates that there exists a minimal faulty schema in this subschemas list of $CandiMFS$, we will update the $CandiMFS$ to be this new minimal faulty schema (line 12), and also update the current minimal faulty schema set $FSS^{\perp}$ (line 13). It is noted that the next location of this minimal faulty schema must be a healthy schema. Hence, we also need to update the current maximal healthy schema set $HSS^{\top}$ (line 14).

With these two auxiliary algorithms, it is easy to describe our complete MFS identification process, which is listed at Algorithm 4.

The input of this algorithm is a failing test case t, and the output is the MFS in this test case. Variable $HSS^{\top}$ records the current maximal healthy schema set, and is initially assigned to be empty (line 1). Variable $FSS^{\perp}$ records the current minimal faulty schema set, and initially have one schema, i.e., test case t.

This algorithm also consists an outer loop (line 3 - 17). In this loop, we first obtain the candidate maximal pending schema set Cxs (line 4) and the candidate minimal pending schema set Cns (line 5). Then a list of ordered pending schema is obtained (line 6). Similar to Algorithm 3, $list = \{c_1, c_2, c_3, ..., c_m\}$, where $c_i \prec c_{i+1}, c_i \in Cns, c_i \in Cxs$, and at each iteration, only the list with the maximal number of pending schemas is select. If we cannot obtain such list of pending schemas (line 7), which indicates that all the schemas in the test case are classified into healthy or faulty, we will break the outer loop (line 8), and return the identified MFS (line 18). Otherwise, we will start the Algorithm 2 to check all the schemas in this list (line 10). If there is no faulty schema in this list (line 11), we only need to update

Algorithm 4 MFS identification

Input: t ▷ failing test case
Output: MFS ▷ MFS in test case t

1: $HSS^{\top} \leftarrow \{\}$
2: $FSS^{\perp} \leftarrow \{t\}$
3: **while** *true* **do**
4: $Cxs \leftarrow CMXS(FSS^{\perp}, t)$
5: $Cns \leftarrow CMNS(HSS^{\top}, t)$
6: $list \leftarrow getLongest(Cxs, Cns, -)$
7: **if** $list == Null$ **then**
8: *break*
9: **end if**
10: $fsIndex \leftarrow Binary\text{-}Search\text{-}Check(list)$
11: **if** $fsIndex == -1$ **then**
12: $HSS^{\top} \leftarrow update(HSS^{\top}, list.get(0))$
13: **else**
14: $fs \leftarrow list.get(fsIndex)$
15: $Narrow\text{-}down\text{-}fs(fs, t, HSS^{\top}, FSS^{\perp})$
16: **end if**
17: **end while**
18: **return** $FSS^{\perp}$

the current maximal healthy schema set (line 12); otherwise, we need to start Algorithm 3 to narrow down the minimal faulty schema in the list (line 14), and update the current minimal faulty schema set (line 15).

To better illustrate this algorithm, we offer an example. Assume there is a failing test case {(p_1, 1), (p_2, 1), (p_3, 1), (p_4, 1), (p_5, 1), (p_6, 1), (p_7, 1), (p_8, 1)}, and there are two MFS in this failing test case, which are {(p_2, 1),(p_3, 1)}, {(p_1, 1), (p_2, 1)}, respectively. The process of our algorithm can be depicted in Fig 1 and Fig 2, in which Fig 1 shows the details of the identification of MFS {(p_2, 1),(p_3, 1)}, while Fig 2 gives an overview of the process for identification both two MFS.

In Fig 1, we can learn that the current minimal faulty schema set $FSS^{\perp}$ is initially have one schema, i.e., test case {(p_1, 1), (p_2, 1), (p_3, 1), (p_4, 1), (p_5, 1), (p_6, 1), (p_7, 1), (p_8, 1)}, and the current maximal healthy schema set $HSS^{\top}$ is initially assigned to be empty. Then we will obtain a list of ordered pending schemas. Note that each schema in this set is neither the subschema of any healthy schema nor the super schema of any faulty schema. After this, we start the Algorithm 2 to use binary search strategy to check the schema in this list. In this checking process, only four schemas need to be checked, and the schema {(p_2, 1), (p_3, 1)} is found to be minimal faulty schema in this list.

The next step is to go on narrowing down this faulty schema {(p_2, 1), (p_3, 1)}. In this step, $FSS^{\perp}$ is updated to be {(p_2, 1), (p_3, 1)}, and $HSS^{\top}$ is updated to be {(p_2, 1)}. We will still need to obtain a list of pending schemas, in which each schema is the subschema of {(p_2, 1), (p_3, 1)}. Finally, only {(p_3, 1)} satisfies this requirement. Next we also need to start the binary search process to check the schema in this list. As there exists only one schema in this list, we only need to check this schema. Up to now, the faulty schema {(p_2, 1), (p_3, 1)} cannot be narrowed down further, because none of its sub schema is faulty schema. Hence, we can learn that schema {(p_2, 1), (p_3, 1)} is one MFS in this failing test case.

We list a example in Fig 2.

3.3 Checking a pending schema

In this section, we will discuss how to check a pending schema.

ASSUMPTION 3. *Given a failing test case t, when we identify the MFS in t, any newly generated test case will not introduce new MFS that is not in t.*

There are similar assumptions given in [25, 11, 12], and we will weaken it later. Based on this assumption, we can get the following lemma which helps to determine whether a pending schema is a healthy schema or a faulty schema.

LEMMA 1. *For a pending schema, we generate an extra test case that contains this schema. If the extra test case passes, then this schema is a healthy schema. If the extra test case fails, then the schema is a faulty schema.*

This lemma is easy to understand, so that we will omit the proof and just offer an example. Assume there is a failing test case {(p_1, 1), (p_2, 1), (p_3, 1), (p_4, 1), (p_5, 1)}, and we need to check the schema {(p_2, 1), (p_4, 1)}. According to Lemma 1, we need to generate additional test case which contains this schema, and the other parameter values in this test case should be different from the original failing test case. Based on this, test case {(p_1, 0), (p_2, 1), (p_3, 0), (p_4, 1), (p_5, 0)} satisfies this requirement (Note that, there may exist other test cases that satisfy this requirement). If this test case fails, then {(p_2, 1), (p_4, 1)} is a faulty schema; otherwise, it is a healthy schema.

3.3.1 Alleviation of assumption 3

If assumption 3 does not hold, it may lead to incorrect pending schema checking. Specifically, if the extra test case fails during testing, we cannot ensure that it is because that the pending schema is a faulty schema, or because that this extra test case contains another MFS which is not contained in the original failing test case. Consequently, we may determine a pending schema to be faulty schema while it is actually a healthy schema.

To alleviate this problem, we augment our MFS identification approach with a feedback mechanism. In detail, after we obtain the MFS with Algorithm 1, we will generate an additional distinct test case which contains the MFS, and execute it. If this test case fails, we stop the whole MFS identification procedure, and regard these schemas identified by Algorithm 1 as MFS; otherwise, we need to re-start Algorithm 1, because the MFS we identified is obviously incorrect. Note that when we re-run Algorithm 1, we can re-use some information from the test cases generated and executed at the previous iterations.

3.4 The number of additional test cases

The number of generated additional test cases is an important metric to measure the cost of the adaptive MFS identification approaches. The test cases generated in our algorithm are used to check the pending schemas. Specifically, in our algorithm, there are two parts may generate additional test cases. First, use *Binary-Search-Check* to find a faulty schema (See line 11 in Algorithm 4); second, use *Narrow-down-fs* to find one minimal faulty schema (See line 16 of Algorithm 4).

Note that when given a list of ordered pending schemas, the number of test cases for the first part is no more than

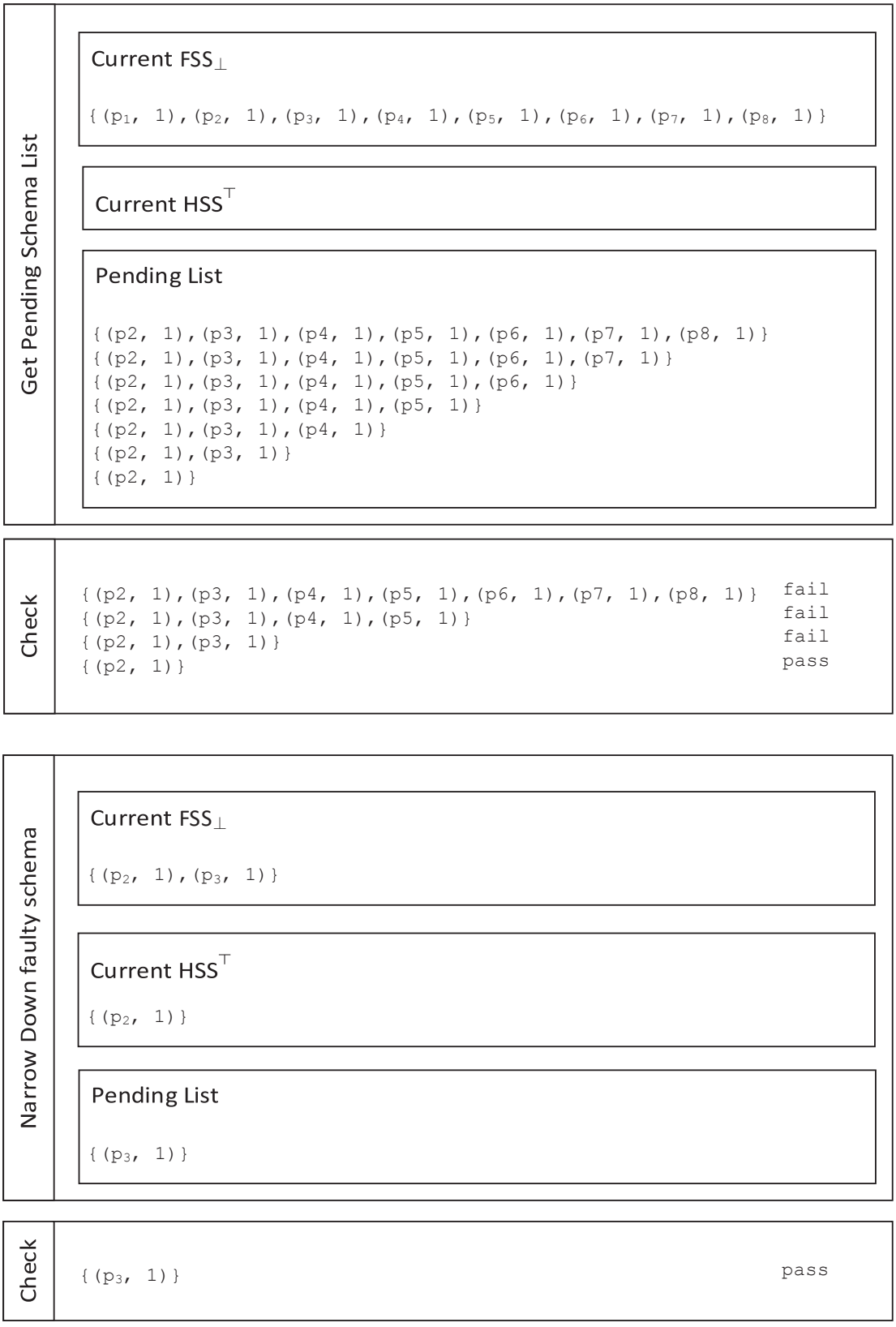

Figure 1: The process of identification of $\{(p_2, 1),(p_3, 1)\}$

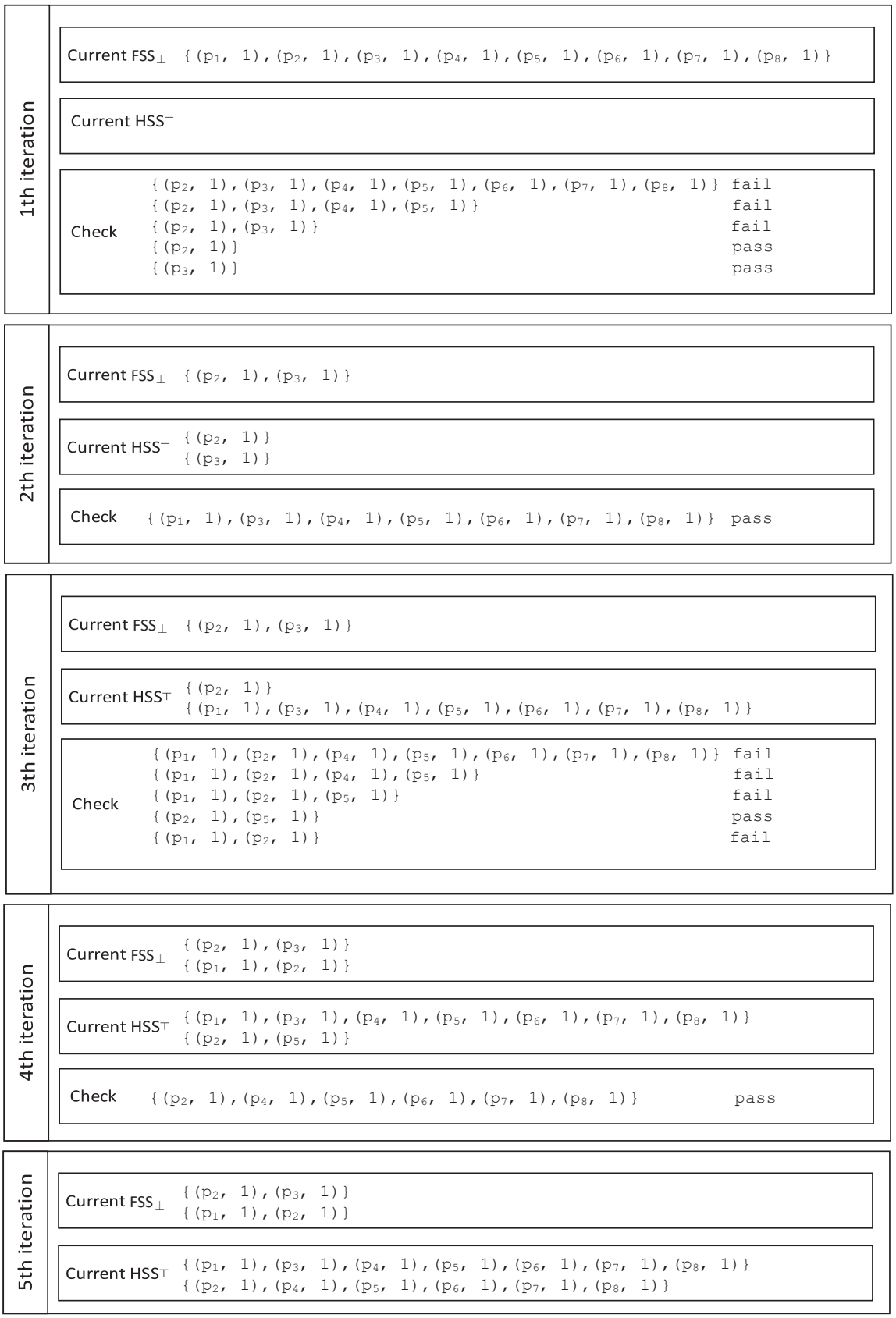

Figure 2: The complete process of identification of two MFS

$log_2 n$, where n is the number of parameters in the SUT (The list obviously consists no more than n pending schemas).

With respect to the second part, it is easy to observe that *Narrow-down-fs* repeatedly call *Binary-Search-Check*. Each time it calls call *Binary-Search-Check*, it actually isolates one failure-inducing factor. This is because *Binary-Search-Check* return the index of the minimal faulty schema in the ordered list of schema, and the schema that is next to this minimal faulty schema is a healthy schema. Note that two schemas that are next to each other only have one parameter value that is different. For example, considering the first pending list in Fig 1, the consecutive schemas $\{(p_2, 1), (p_3, 1)\}$ and $\{(p_2, 1)\}$ only have one parameter value $\{(p_3, 1)\}$ that is different. Based on this, the difference between the minimal faulty schema and the heathy schema is only one parameter value, which must be failure-inducing factor according to Nie's work [14]. Consequently, for the second part *Narrow-down-fs*, it needs to call *Binary-Search-Check* at most $\tau - 1$ times to obtain the MFS, where τ is the degree of the MFS. Note that here the iteration number is not τ, because before we execute the second part *Narrow-down-fs*, *Binary-Search-Check* has already execute once in the first part, and one failure-inducing factor must has already be located. Hence, the number of test cases that are needed by the second part is no more than $(\tau - 1) \times log_2 n$.

Next we need to consider how many times will these two parts be executed. For the second part, it needs to execute k times, where k is the number of MFS in the failing test case. This is because, each time *Narrow-down-fs* is called, it will finally return a MFS. Hence, there needs $k \times (\tau - 1) \times log_2 n$

With respect to the first part, besides k times that it is needed to be called to detect the faulty schemas, it still needs more test cases to ensure that the remaining schemas are not faulty. This number is no more than the number of schemas in $CMXS(FSS^{\perp}, t)$, in which the schemas in this set are candidate maximal pending schemas. According to the definition of CMXS, this number is no more than τ^k, where τ is the degree of the MFS. As these schemas are healthy schemas, *Binary-Search-Check* function only needs 1 additional test case (Only check the first schema in the list). Finally, there is $k \times log_2 n + \tau^k$ in total for the second part.

At last, the number of test cases generated by the first part and second part is no more than $k \times (\tau - 1) \times log_2 n + k \times log_2 n + \tau^k = k \times \tau \times log_2 n + \tau^k$. Append the additional test cases for re-check each MFS (For alleviation of assumption 3), the total number needed by our algorithm is no more than $k \times \tau \times log_2 n + \tau^k + k$.

4. WEAKEN ASSUMPTIONS 1 AND 2

In this section we will discuss the impacts on MFS identification of the two assumptions proposed in Section 2, as well as how to weaken them.

The first assumption is that the outcomes of all the tests are deterministic. In practice, re-executing the same test case may result in different outcomes. For example, if the program using some random variables, different runs of the test case will assign different values to these random variables. Non-deterministic results will complicate the MFS identification, and even worse, it may lead to an unreliable result of the MFS identification. Inspired by the idea that using multiple same-way covering arrays to identify the MFS [3, 21], one potential solution to alleviate this non-deterministic failure is by adding redundancy, i.e., through re-executing the same test case to obtain the relatively stable outcome.

The second assumption is that if a test case contains a MFS, it must fail as expected. Some effects [13, 22] may make this assumption invalid. According to the study [25], we can learn that the schemas that are identified under this condition may be the super-schema of actual MFS. Hence, in practice, we may not only need to check the schemas that are identified as MFS, but also need to pay attention to their sub-schemas.

5. EVALUATION WITH SIMULATED SUT

In this section, we designed a simulated SUT of which the number of parameters and the value of each parameter can both be customized. In addition, we can also manually inject faulty schemas in the simulated SUT. We will conduct a series of experiments with this simulated SUT. The goal of our experiments is to evaluate the efficiency and effectiveness of our approach when compared with other existing techniques. The main reason that we use simulated program is that we can easily run a set of experiments with various states of a SUT, i.e., different parameters and different MFSs. By doing so we can thoroughly learn the performance of each algorithm without biases.

5.1 Approaches for comparison

There are several approaches that aim at identifying the MFS, which can be classified as non-adaptive methods and adaptive methods. The former ones do not need additional test cases while the latter do. Our algorithm belongs to the adaptive methods. To make the comparison clear and fair, we select all the adaptive MFS identification in this study, which are listed as follows:

(1) OFOT [14] – It changes one factor of a test case one time to generate a new test case and then analyse the MFSs according to the executed result of these test cases. (2) FIC_BS [25] – It uses a delta debugging strategy to isolate the MFSs (3) LG [11, 12] – It uses a locatable graph to represent the faulty schema and adopts the divide and conquer strategy to find the MFSs. (4) SP[5] – It generates test cases for the most suspicious schemas and gives a rank of these schemas. (5) CTA[17], it combines the OFOT approach and the classified tree technique which is firstly applied in research [21] to analyse the MFSs. (6) RI [10] – It also uses delta debugging, but mutates a different factors when compared with FIC_BS. (7) AIFL [20] – It extends OFOT, which changes different number of factors instead of one at each iteration for a test case. (8) TRT [16] and (9) TRTNA [16] – They use a structured tree model to analyse the MFSs in a failing test case. More details of these algorithms will be discussed in the related works. (10) Furthermore, our approach in this comparison will be denoted as CMS, which represents Candidate Minimal/Maximal pending Schemas.

With respect to *SP*, the degree of MFS set for this algorithm is 2 (except for the last group of experiment, which is 4 instead). We list a overview of these approaches in Table 6. In this table, Column "Tests" lists the number of additional test cases generated by each approach. These number vary from different approaches, and are related to the number of parameters, i.e., n, in the SUT, the number of MFS, i.e., k, to be identified, and the degree of MFS, i.e., τ. Column "Degree" shows the degree of MFS that each

approach can identify. We can learn that almost all the approaches can identify the MFS with degree ranges from 1 to n, except approach "LG", which can only identify the MFS with degree 2, and approach SP, which can only identify the MFS with degree no more than a given number t. Column "Multiple" indicates that whether the approach can handle the condition that a failing test case contain multiple MFS. Result "Multiple" shows that the corresponding approach can handle such condition, while "Single" indicates that the approach cannot handle this type of test case. Note that some approaches can only handle the condition that multiple MFS have no overlapped part (has no same parameter value). These approaches will be labeled with "none overlapped". At last, Column "MFS introduce" shows whether the approach can deal with the newly introduced MFS in the additional generated test cases. Signal ⊘ means that the corresponding approach cannot handle such case, while ⊙ indicates that the approach is able to deal with it.

5.2 Evaluation metrics

There are two focuses in this study: 1)How many additional test cases do these approaches need to generate? 2) The effectiveness of these approaches? We apply recall and precision to evaluate the second question :

$$recall = \frac{correctly\ Identified\ MFS}{all\ the\ MFS\ in\ SUT}$$

$$precision = \frac{correctly\ Identified\ MFS}{all\ the\ identified\ MFS}$$

Recall measures how well the real MFS are detected and identified. *Precision* shows the degree of accuracy of the identified schemas when comparing to the real MFS.

5.3 Experiment setups

We will carry out the following five experiments in this section:

1)For the first one, we will give a set of SUTs with 8 parameters, each parameter has 3 values (3^8). We inject each of them with only one distinct 2-degree MFS. There are $\binom{8}{2} = 28$ possible MFSs we can inject for a test case with 8 parameters, hence, the number of SUTs in this study is 28. For each SUT in this set, we will feed each approach a failing test case with the MFS contained in it, and then let these approaches identify the MFS. We will record the additional test cases each approach needed, as well as the precision and recall with respect to the identified MFS.

2)The second experiment is similar to the first one, except that we will inject two different 2-degree MFSs in each SUT. It is easily compute that there are $\binom{28}{2} = 278$ possible SUTs in this experiment. We also feed these approaches with a failing test case that contains the injected two MFSs and then let them identify the MFS.

3)The third experiment aims at measuring capability of each algorithm at handling the newly introduced MFSs. To accomplish this goal, We firstly inject two MFSs in a SUT, and then feed each approach a failing test case which contains only one 2-degree MFS, with another 1-degree MFS not in it (Note that 1-degree MFS can be easily introduced by just changing one factor of the original failing test case). There are $\binom{8}{2} \times \binom{8}{1} = 224$ possible SUTs in this experiment.

4)For the fourth experiment, we will take 8, 9, 10, 12, 15, 20, 30, 40, 60, 80, 100, and 120, respectively, as the number of parameters of SUT. For each number of parameters, say n, we will generate $\binom{n}{2}$ SUTs, each of which will be injected a single distinct 2-degree MFS. For these SUT, we will follow the same process as experiment 1. We will record the average number of additional test cases that each approach needed for each number of parameters. The goal of this experiment is to determine the influence of the number of parameters of SUT on each approach.

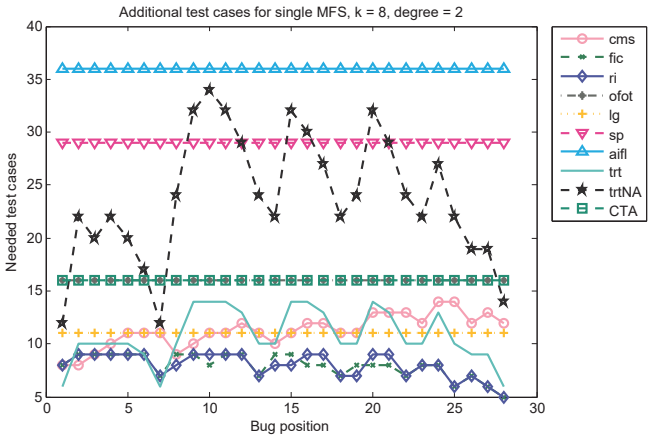

Figure 3: k = 8, t = 2,single, additional test suites

5)In the fifth experiment we will fix the number of parameters of SUT to be 8, but vary the number of degrees of the MFS that we inject in the SUT. That is, we will inject the MFS in SUT with degrees 1, 2, 3, 4, 5, 6, 7, and 8, respectively. For a degree of m (m = 1, 2, 3, 4, 5, 6, 7, and 8), we will generate $\binom{8}{m}$ SUTs with different m-degree MFS injected. Then we will let all those approaches identify the MFS in them. This experiment helps to inspect whether these approaches can handle different degrees of MFS in a failing test case.

5.4 Result and discussion

1)The results of the experiment 1 are depicted in figure 3. The y-axis represents the number of test cases and the x-axis represents SUT with different MFS injected. The polygonal lines represent the results for each corresponding approach. For a particular approach, a point in its polygonal line indicates the number of test cases (y-axis) needed for identifying the MFS in the SUT (x-axis).

There are some observations we can learn from this figure. First, the test cases that are needed of some approaches did not change along with the MFS we injected. These approaches are AIFL, SP, CTA, OFOT, and LG, respectively, while others need different test cases according to the MFS that they need to identify. Second, some approaches (AIFL, SP, TRTNA, OFOT, CTA) showed an obvious disadvantage, they need much more test cases than the remaining approaches. For the remaining approaches, the disparity of their results is not significant. From a deeper insight of this figure, we can conclude an initial rank in the number of test cases each approach needed as: $FIC\_BS < RI < LG < TRT < CMS < OFOT = CTA < TRTNA < AIFL < SP$.

With respect to the precision and recall, all these approaches obtain the same performance (both precision and recall are 1). This is because the SUT with single 2-degree

Table 6: The overview of each algorithm

Approaches	Tests	Degree	Multiple	MFS introduce
OFOT	n	1∼n	single	⊘
FIC_BS	$\leq k \times (\tau \times (log_2(n)+1))$	1∼n	multiple, none overlapped	⊘
LG	$k \times log_2(n) + k^2$	2	multiple	⊘
SP	$\geq$ n	$\leq$ d	multiple	⊙
CTA	n	1∼n	single	⊘
RI	$k \times (\tau \times (log_2(n)+1))$	1∼n	multiple, none overlapped	⊘
AIFL	$\leq 2^n$	1∼n	multiple, none overlapped	⊘
TRT	$k \times \tau \times log_2 n + \tau^k$	1∼n	multiple	⊘
TRTNA	$3(\times k \times \tau \times log_2 n + \tau^k)$	1∼n	multiple	⊙
CMS	$k \times \tau \times log_2 n + \tau^k + k$	1∼n	multiple	⊙

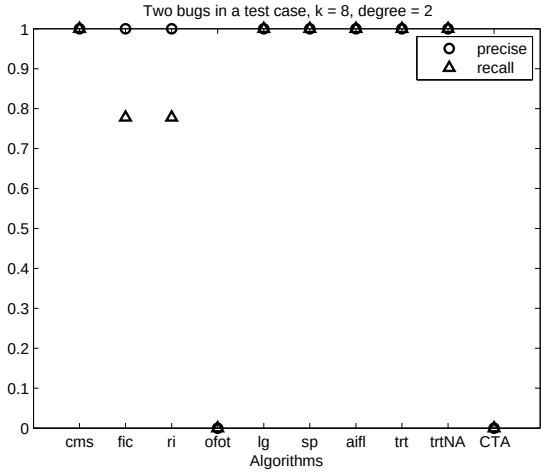

Figure 4: k = 8, t = 2,double, additional test suites

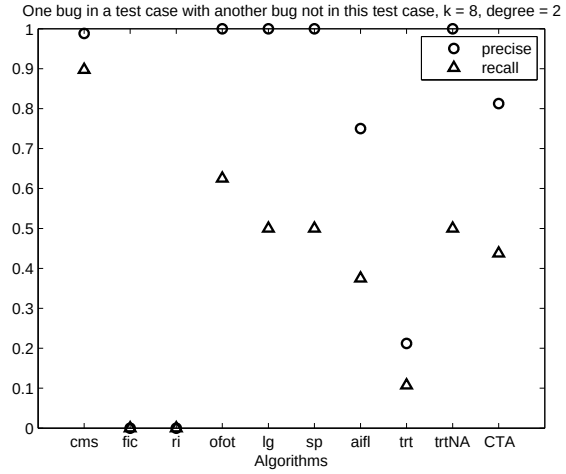

Figure 5: k = 8, t = 2,import, additional test suites

MFS is a very simple scenario, at which all these approaches can identify the MFS.

2)Figure 4 gives the results of experiment 2. This figure does not show the number of test cases of each approach generated, instead it just shows the average precision and recall for each approach as not all the approaches can both get recall of 1 and precision of 1 as experiment 1. It is obvious meaningless if we compare two approaches, when one of them can obtain recall of 1 and precision of 1 while the other can not.

From Figure 4, we can find the following approaches: CMS, LG, SP, AIFL, TRT and TRTNA can obtain 1 for both precision and recall, which means that they can properly handle the multiple MFSs in a failing test case. In the opposite case, approaches OFOT and CTA get 0 for both recall and precision. This result shows that these two approaches can't handle multiple MFSs in one failing test case. As for the remaining approaches RI and FIC_BS, they obtained precision of 1, but did not obtain recall of 1. This is because they can't handle the condition when two MFSs have overlapped part. Under that condition, they can identify only one MFS.

3)The results of the third experiment are shown in fig 5. Similar to experiment 2, this figure just records the recall and precision of each approach. There are several observations in this figure.

First, no approach can obtain 1 for both recall and precision in this experiment, which indicates that no approach can fully resolve the new MFS introducing problem.

Second, there are some approaches, i.e., OFOT, LG, SP, TRTNA, and our approach CMS, that can obtain precision of 1. It indicates that the newly introduced MFS has little impact on the accuracy of MFS they identified.

Third, our approach CMS get the highest score at recall, and the specific rank are $CMS > OFOT > LG = SP = TRTNA > CTA > AIFL > TRT > FIC\_BS = RI$. It shows that our approach can find more MFSs than others when just given a failing test case.

Fourth, approaches FIC_BS and RI really suffer from the new MFS introducing problem, because they get 0 at both recall and precision.

4)Figure 6 depicts that the average test cases needed of each approach for experiment 4. In this figure, the y-axis represents the number of test cases, and the x-axis represents the number of parameters of the SUT. The number of parameters are 8, 9, 10, 12, 15, 20, 30, 40, 60, 80, 100, and 120, respectively. A point in this figure indicates the average number of test cases needed to identify the MFS in the SUT with the corresponding number of the parameters. Note that in this figure, the average numbers of test cases of approaches SP, AIFL, OFOT and CTA increase quickly with the increase of k. It could make the results of the remaining approaches unclear if we put all of them in the same figure. Hence, we just show the results of the remaining approaches

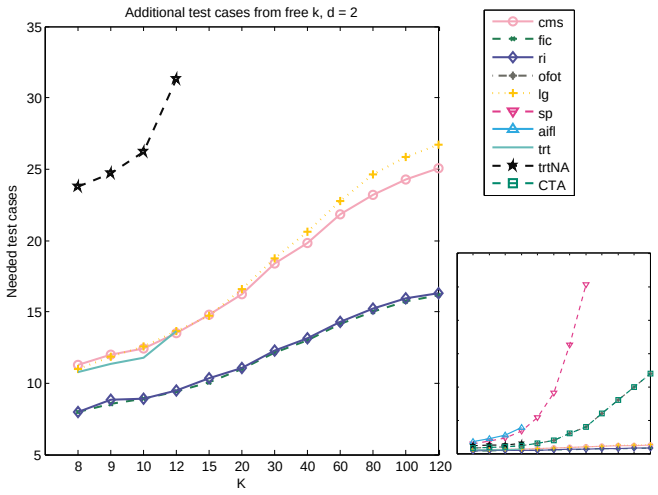

Figure 6: k is free, t = 2,single, additional test suites

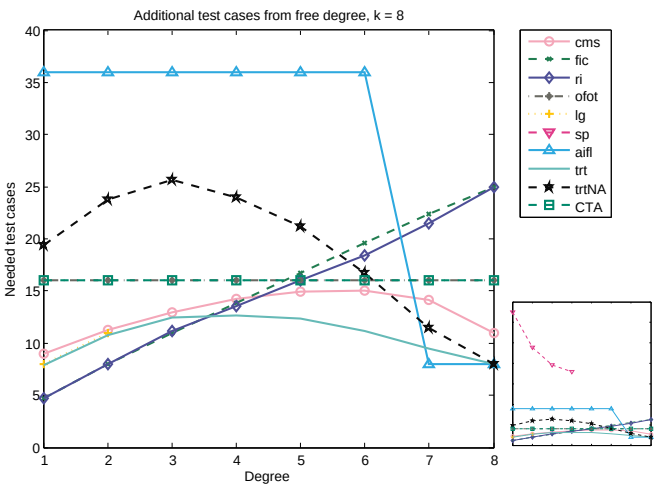

Figure 7: k = 8, t is free,single, additional test suites

(TRTNA, TRT, LG, CMS , FIC_BS, and RI), and offer a smaller figure which shows the results of all the approaches.

We can easily learn that AIFL needs the largest number of test cases to identify MFS. Even worse, the computing resource that it needs also increases quickly along with the increase of k. Consequently we cannot use this approach to identify the MFS in the SUT with number of parameters larger than 12. Approach SP performs the second worst in this experiment. In fact, we cannot record its results when the number of parameters is greater than 40. OFOT and CTA are the third worst approaches in this experiment, with the number of test cases increases linearly with the increase of k.

The remaining approaches need much smaller number of test cases than four previously discussed approaches. Among these approaches, however, TRT and TRTNA cannot identify the MFS in the SUT with k greater than 12 (Due to their high computing complexity). Apart from this two approaches, all the remaining approaches, i.e., CMS, FIC_BS, LG, and RI can quickly identify the MFS for different values of k. We can conclude a rank of these approaches according to the number of test cases generated, $AIFL > SP > TRTNA > OFOT = CTA > LG > CMS > TRT > RI > FIC\_BS$.

5)Figure 7 depicts the results of the last experiment. This figure is organised similarly as figure 6, except that the x-axis in this figure represents the degree of MFS we inject in it. For a particular approach, a point in this figure shows the average number of test cases needed to identify the MFS. Note that in this experiment we set the degree of MFS that SP can identify to be 4 (In this experiment, SP can only identify the MFS with degree no more than 4). As a result, the number of test cases that SP need to identify the MFS in a failing test cases increases a lot than before, which is much more than the remaining approaches. Hence, we first show the results of the remaining approaches, and offer a smaller figure which added the results of SP.

Some observations in this figure includes: First, LG can only identify the MFS with degree no more than 2. Second, we can find the numbers of test cases of RI and FIC_BS increase quickly along with the increase of degree. Third, CTA and OFOT need the same number of test cases, i.e., 16, regardless of what the degree is. Fourth, although AIFL perform the worst in most cases, it can do better than other approaches under a high degree (7 and 8). Fifth, the numbers of test cases needed by TRTNA, TRT and our approach CMS increase slowly with the increase of degree.

In summary, our approach obtains good values at recall and precision with respect to the MFS identification, while generates much smaller number of test cases than most existing approaches. Furthermore, our approach performs better than most existing approaches under the following conditions: multiple MFS in one test case (Especially the MFS having overlapped part), new MFS introducing problem, SUT with a large number of parameters, and MFS with a high degree.

6. EMPIRICAL CASE STUDY

In this section, we will conduct the empirical study on several real-life software subjects.

6.1 Subject programs

The subject programs used in our experiments are listed in Table 7. Column "Subjects" indicates the specific software. Column "Version" indicates the specific version that is used in the following experiments. Column "LOC" shows the number of source code lines for each software. Column "Faults" presents the fault ID, which is used as the index to fetch the original fault description from the bug tracker for that software. Column "Lan" shows the programming language for each software (For subjects written in more than one programming language, only the main programming language is shown). We select these software as subjects because their behaviours are influenced by various combinations of configuration options or inputs. For example, the component *connector* of Tomcat is influenced by more than 151 attributes [6].

To conduct this study, we firstly must know the faults and their corresponding MFS in prior, so that we can determine whether the schemas identified by different approaches are accurate or not. For this, we looked through the bug tracker of each software and focused on the bugs which are caused by the interaction of configuration options. Then for each such bug, we obtain its MFS by analysing the bug description

Table 7: Subject programs

Subjects	Version	LOC	Faults	Lan
Tomcat	7.0.40	296138	#55905	java
Hsqldb	2.0rc8	139425	#981	Java
Gcc	4.7.2	2231564	#55459	c
Jflex	1.4.2	10040	#87	Java
Grep	2.12	27046	#7600	c
Cli	1.3.1	6433	#265	java
Joda	2.8.2	85026	#297	java
Lang	3.4	66047	#1205	java
Jsoup	1.8.3	10295	#688	java

report and the associated test file which can reproduce the bug.

6.1.1 *Specific inputs models*

To apply MFS identification approaches on the selected software, we need to firstly model their input parameters. It includes the options that cause the specific faults in Table 7, and some additional options, which are used to create some noise for the MFS identification approach. A brief overview of the input models as well as the corresponding MFS (degree) is shown in Table 8.

Table 8: Inputs model

Subjects	Inputs	MFS
Tomcat	$2^8 \times 3^1 \times 4^1$	1(1) 2(2)
Hsqldb	$2^9 \times 3^2 \times 4^1$	3(2)
Gcc	$2^9 \times 6^1$	3(4)
Jflex	$2^{10} \times 3^2 \times 4^1$	2(1)
Grep	$2^5 \times 3^1 \times 4^1$	2(3)
Cli	2^8	3(1)
Joda	$2^5 \times 3^2 \times 5^1$	2(1)
Lang	$2^4 \times 3^3$	2(1)
Jsoup	$2^8 \times 3^1$	2(6)

In this table, Column "inputs" depicts the input model for each version of the software, presented in the abbreviated form $\#values^{\#number\ of\ parameters} \times ...$, e.g., $2^9 \times 3^2 \times 4^1$ indicates the software has 9 parameters that can take on 2 values, 2 parameters can take on 3 values, and only one parameter that can take on 4 values. Column "MFS" shows the degrees of each MFS and the number of MFS (in the parentheses) with that corresponding degree.

Note that these inputs just indicate the combinations of configuration options. To conduct the experiments, some other files are also needed. For example, besides the XML configuration file, we need a prepared HTML web page and a java program to control the startup of the tomcat to see whether exceptions will be triggered.

6.2 Study setup

For each subject program listed in Table 7, we firstly obtain all of the failing test cases. Then, for each failing test case, we will feed it to each MFS identification approach and let them identify the MFSs. Then we will compare the schemas identified by each approach with the real-life MFSs we obtained in prior. We will record the number of generated test cases, the precision, and the recall. To be fair, no other information is given to each approach except the feeded failing test case. We will repeat this comparison for each failing test case. At last, we will give the average number of test cases, precision and recall for each approach.

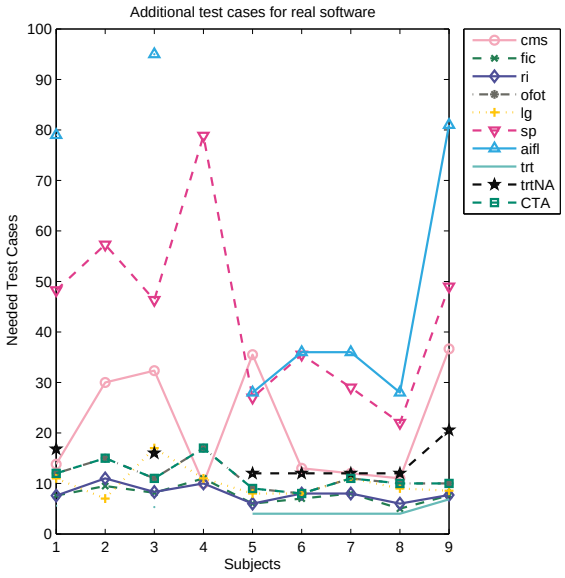

Figure 8: Additional test suites for real-life software subjects

6.3 Results and analysis

Figure 8 depicts the number of generated test cases of each approach. In this figure, the y-axis represents the number of test cases, and the x-axis represents the subject programs. Each point of corresponding approach shows the average test cases generated for identifying the MFS in the SUT. It is noted that for approaches AIFL, TRT, and TRTNA, they cannot identify the MFS in subject 2 and 4 because of their large configuration space.

From this figure, we can learn that approaches AIFL, and SP generated much more test cases than the remaining approaches. Approaches TRT, LG, FIC_BS, needed the smallest number of test cases. The remaining approaches are in between. It is noted that our approach performed good for subjects 1, 4, 6, 7, and 8, but needed many test cases for subjects 2, 3, 5, and 9. We believe this is because for those subjects, there are too many faults (there are 2, 4, 3, and 6 MFSs for these subjects) in their test cases, which may result in many new MFS introducing problem such that our approach repeated checking the results.

The results of precision and recall are shown in Table 9, and Table 10, respectively. In these two Tables, Column "App" list all the approaches, and the remaining 9 columns show the precision and recall obtained by these approaches in this experiment. We emphasize the result in bold if it is the best among all the approaches. We can observe that with respect to precision, there are 6 out of 9 subjects (subjects 2, 4, 5, 6, 7, 8) at which our approach CMS performs the best when compared with others; and with respect to recall, there are also 6 out of 9 subjects (subjects 3, 4, 5, 6, 7, 8) at which CMS is the best. This ratio is the highest among all the MFS identification approaches, which is an evidence that CMS can work well at MFS identification on real-life

Table 9: The precision of approaches for 9 real-life software

App	Precision								
CMS	0.7	**1**	0.58	**1**	**1**	**1**	**1**	**1**	0.27
FIC_BS	0.7	0.5	0.17	**1**	0.33	**1**	**1**	**1**	0.07
RI	0.7	0.5	0.17	**1**	0.33	**1**	**1**	**1**	0.07
OFOT	0.8	**1**	0.67	**1**	0.39	**1**	**1**	**1**	0.43
LG	**1**	0	0	0	0	0	**1**	**1**	0
SP	0.28	0	0	0.62	0	0	**1**	**1**	0.03
AIFL	1	-	0.5	-	0.33	**1**	**1**	**1**	0.43
TRT	**1**	-	**1**	-	**1**	**1**	**1**	**1**	**1**
TRTNA	**1**	-	**1**	-	**1**	**1**	**1**	**1**	**1**
CTA	0	0.67	0.22	1	0.67	**1**	**1**	**1**	0.43

Table 10: The recall of approaches for 9 real-life software

App	Recall								
CMS	0.33	0.75	**0.38**	**1**	**0.67**	**1**	**1**	**1**	0.15
FIC_BS	0.33	0.25	0.04	**1**	0.11	**1**	**1**	**1**	0.01
RI	0.33	0.25	0.04	**1**	0.11	**1**	**1**	**1**	0.01
OFOT	0.33	**1**	0.25	**1**	0.33	**1**	**1**	**1**	0.14
LG	**0.4**	0	0	0	0	0	**1**	**1**	0
SP	0.33	0	0	0.62	0	0	**1**	**1**	0.01
AIFL	**0.4**	-	0.17	-	0.11	**1**	**1**	**1**	0.11
TRT	**0.4**	-	0.33	-	0.33	**1**	**1**	**1**	**0.29**
TRTNA	**0.4**	-	0.33	-	0.33	**1**	**1**	**1**	**0.29**
CTA	0.33	**1**	0.25	**1**	**0.67**	**1**	**1**	**1**	0.14

software subjects.

In summary, our approach performs good at real-life software subjects. It can accurately identify the MFS in these subjects while generates a small number of test cases. This result coincides with the result of experiments conducted on simulated subjects.

6.4 Threats to validity

There are several threats to validity in our empirical studies. First, our experiments are based on 9 open-source software. More subject programs are desired to make the results more general. Second, to some extent the results of our approach depend on the generated test cases for checking an interaction, i.e., if the test cases are chosen properly, it may reduce the possibility that a new MFS is introduced, and our approach can reduce test cases for repeatedly checking whether our results are correct or not. In this paper, we just simply select one test case that is different from the original failing test case; more researches may be needed to study how to select test cases for checking pending schemas.

7. RELATED WORKS AND DISCUSSION

Combinatorial testing has been widely applied in industry [7]. Fault localization is an important part of CT studies. [15], as it can facilitate debugging efforts by reducing the scope of code that is needed for inspection [5].

Shi and Nie [18] presented an approach for failure revealing and failure diagnosis in CT , which first tests the SUT with a covering array, then reduces the value schemas contained in the failing test case by eliminating those appearing in the passing test cases.

Nie's approach in [14] first separates the faulty-possible tuples and healthy-possible tuples into two sets. Subsequently, by changing one parameter value at a time of the original test case, this approach generates extra test cases. After executing the configurations, the approach converges by reducing the number of tuples in the faulty-possible sets. Wang then [20] extended this approach by adaptively mutates multiple parameter values instead of only one factor at a time.

Delta debugging [23] proposed by Zeller is an adaptive divide-and-conquer approach to locate interaction fault. It is very efficient and has been applied to real-life software environment. Zhang et al. [25] also proposed a similar approach that can identify the failure-inducing combinations that have no overlapped part efficiently. JieLi proposed a similar approach [10].

Charles [2] proposed the notion of (d,t)-detecting array, which corresponds to test cases that permit the determination of MFS in the SUT, if the number of MFS d and degree of MFS t are known in prior. C. Martiez [11] also proposed a non-adaptive method. Their approach extends the covering array to the locating array to detect and locate interaction faults. They [12] then extend this non-adaptive method to any degree with the theory of hyber-graphic. C. Martiez [11, 12]proposed two adaptive algorithms. The first one needs safe value as their assumption and the second one remove this assumption when the number of values of each parameter is equal to 2. Their algorithms focus on identifying the faulty tuples that have no more than 2 parameters.

Ghandehari.etc [5] defines the suspiciousness of tuple and suspiciousness of the environment of a tuple. Based on this, they rank the possible tuples and generate the test cases. Although their approach imposes minimal assumption, it does not ensure that the tuples ranked in the top are the faulty tuples. They further utilized the test cases generated from the inducing interaction to locate the fault [4].

Yilmaz [21] proposed a machine learning method to iden-

tify inducing combinations from a combinatorial testing set. They construct a classified tree to analyze the covering arrays and detect potential faulty combinations. They [21] additionally extend their work on variable covering array. Beside this, Fouché [3] and Shakya [17] made some improvements in identifying failure-inducing combinations based on Yilmaz' work.

Zhang [24] proposed an approach that are directly based on covering array. This approach translates MFS identification to a constraints satisfaction and optimal problem.

Our previous work [16] 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.

8. CONCLUSION

In this paper, we proposed a new approach to identify failure-inducing interactions. It considers many issues that may negatively influence the performance of MFS identification while at the same time consuming small number of test cases. We further conducted a series of empirical studies based on both synthetic and real-life software subjects. These studies comprehensively compare the performance of all the existing adaptive MFS identification approaches. The results exhibit that our approach can obtain a very good efficiency and effectiveness when compared with other approaches.

As a future work, we would like to analyse the relationship between the failure-inducing interactions and the actual faulty code which cause the failure, so that we can better apply the Combinatorial testing on code-level fault localization. Another interesting work lies on the application of our approach. It is appealing to apply our approach to identify the MFS on more industrial software systems.

9. REFERENCES

[1] J. Bach and P. Schroeder. Pairwise testing: A best practice that isnt. In *Proceedings of 22nd Pacific Northwest Software Quality Conference*, pages 180–196. Citeseer, 2004.

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

[3] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 177–188. ACM, 2009.

[4] L. S. Ghandehari, Y. Lei, D. Kung, R. Kacker, and R. Kuhn. Fault localization based on failure-inducing combinations. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 168–177. IEEE, 2013.

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

[6] K. Kolinko. The HTTP Connector. http://tomcat.apache.org/tomcat-7.0-doc/config/http.html, 2014. [Online; accessed 3-Nov-2014].

[7] D. R. Kuhn, R. N. Kacker, and Y. Lei. Practical combinatorial testing. *NIST Special Publication*, 800:142, 2010.

[8] D. R. Kuhn and M. J. Reilly. An investigation of the applicability of design of experiments to software testing. In *Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE*, pages 91–95. IEEE, 2002.

[9] D. R. Kuhn, D. R. Wallace, and J. AM Gallo. Software fault interactions and implications for software testing. *Software Engineering, IEEE Transactions on*, 30(6):418–421, 2004.

[10] J. Li, C. Nie, and Y. Lei. Improved delta debugging based on combinatorial testing. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 102–105. IEEE, 2012.

[11] C. Martínez, L. Moura, D. Panario, and B. Stevens. Algorithms to locate errors using covering arrays. In *LATIN 2008: Theoretical Informatics*, pages 504–519. Springer, 2008.

[12] C. Martínez, L. Moura, D. Panario, and B. Stevens. Locating errors using elas, covering arrays, and adaptive testing algorithms. *SIAM Journal on Discrete Mathematics*, 23(4):1776–1799, 2009.

[13] W. Masri and R. A. Assi. Prevalence of coincidental correctness and mitigation of its impact on fault localization. *ACM Trans. Softw. Eng. Methodol.*, 23(1):8:1–8:28, Feb. 2014.

[14] C. Nie and H. Leung. The minimal failure-causing schema of combinatorial testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):15, 2011.

[15] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11, 2011.

[16] X. Niu, C. Nie, Y. Lei, and A. T. Chan. Identifying failure-inducing combinations using tuple relationship. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 271–280. IEEE, 2013.

[17] K. Shakya, T. Xie, N. Li, Y. Lei, R. Kacker, and R. Kuhn. 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*, pages 620–623. IEEE, 2012.

[18] L. Shi, C. Nie, and B. Xu. A software debugging method based on pairwise testing. In *Computational Science–ICCS 2005*, pages 1088–1091. Springer, 2005.

[19] C. Song, A. Porter, and J. S. Foster. itree: Efficiently discovering high-coverage configurations using interaction trees. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 903–913. IEEE Press, 2012.

[20] Z. Wang, B. Xu, L. Chen, and L. Xu. Adaptive interaction fault location based on combinatorial testing. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 495–502. IEEE, 2010.

[21] C. Yilmaz, M. B. Cohen, and A. A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Software Engineering, IEEE Transactions on*, 32(1):20–34, 2006.

[22] C. Yilmaz, E. Dumlu, M. Cohen, and A. Porter. Reducing masking effects in combinatorialinteraction testing: A feedback drivenadaptive approach. *Software Engineering, IEEE Transactions on*, 40(1):43–66, Jan 2014.

[23] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.

[24] J. Zhang, F. Ma, and Z. Zhang. Faulty interaction identification via constraint solving and optimization. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 186–199. Springer, 2012.

[25] Z. Zhang and J. Zhang. Characterizing failure-causing parameter interactions by adaptive testing. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 331–341. ACM, 2011.

APPENDIX

We will give the proofs of several important propositions.

Proposition 5.

PROOF. Let A = $\{c \mid c \prec t \;\&\&\; c_1 \not\preceq c\}$, and B = $\{c \mid \exists c_1' \in CMXS(c_1,t),\ c \preceq c_1'\}$.

Let $c_1 = \{(p_{x_1}, v_{x_1}), (p_{x_2}, v_{x_2}), ..., (p_{x_k}, v_{x_k})\}$, t = $\{(p_1, v_1), (p_2, v_2), ..., (p_n, v_n)\}$, and CMXS($c_1$,t)= $\{t\backslash(p_{x_i}, v_{x_i}) \mid (p_{x_i}, v_{x_i}) \in c_1\}$.

First we will show A ⊆ B.

With respect to set A, $\forall c' \in A$, it has $c' \prec t$ and $c_1 \not\preceq c'$. That is, $\forall e \in c', e \in t$, and $\exists e' \in c_1, e' \notin c'$. As $c_1 \preceq t$, $e' \in t$. Hence, we have $\forall e \in c', e \in t\backslash e'$, i.e., $c' \preceq t\backslash e'$.

Since $t\backslash e' \in CMXS(c_1,t)$, $c' \in \{c \mid \exists c_1' \in CMXS(c_1,t), c \preceq c_1'\}$ = B. Hence, A ⊆ B.

Second we will show B ⊆ A.

With respect to set B, $\forall c' \in B$, it has $\exists c_1' \in CMXS(c_1,t), c' \preceq c_1'$. Since $c_1' \in CMXS(c_1,t)$, $\exists e' \in c_1, c_1' = t\backslash e'$. Consequently, $c' \preceq t\backslash e'$. Hence, $c_1 \not\preceq c'$. Also, $c' \preceq t\backslash e' \prec t$. Consequently, $c \in \{c \mid c \prec t \;\&\&\; c_1 \not\preceq c\}$ = A, which indicates that B ⊆ A.

As we have shown B ⊆ A, and A ⊆ B, so A = B.

□

Proposition 6.

PROOF. We just need to prove that for two faulty schemas c_1, c_2, and a failing test case t ($c_1 \preceq t$, $c_2 \preceq t$), we have $\{c \mid c \prec t \;\&\&\; \forall c_i \in \{c_1, c_2\}, c_i \not\preceq c\} = \{c \mid \exists c_1' \in CMXS(c_1,t) \bigwedge CMXS(c_2,t),\ c \preceq c_1'\}$.

Let $A = \{c \mid c \prec t \;\&\&\; \forall c_i \in \{c_1, c_2\}, c_i \not\preceq c\}$, $A_1 = \{c \mid c \prec t \;\&\&\; c_1 \not\preceq c\}$, $A_2 = \{c \mid c \prec t \;\&\&\; c_2 \not\preceq c\}$. It is easily to get $A = A_1 \bigcap A_2$.

Let $B = \{c \mid \exists c_1' \in CMXS(c_1,t) \bigwedge CMXS(c_2,t),\ c \preceq c_1'\}$. Here, $CMXS(c_1,t) \bigwedge CMXS(c_2,t) = \{c \mid c = c_1' \cap c_2',\ where\ c_1' \in CMXS(c_1,t),\ and\ c_2' \in CMXS(c_2,t)\}$.

Let $B_1 = \{c \mid \exists c_1' \in CMXS(c_1,t),\ c \preceq c_1'\}$, and $B_2 = \{c \mid \exists c_2' \in CMXS(c_2,t),\ c \preceq c_2'\}$. $B_1 \bigcap B_2 = \{c \mid \exists c_1' \in CMXS(c_1,t),\ c \preceq c_1' \;\&\&\; \exists c_2' \in CMXS(c_2,t),\ c \preceq c_2'\}$. Note that, $c \preceq c_1' \;\&\&\; c \preceq c_2' \equiv c \preceq c_1' \cap c_2'$. Hence, $B_1 \bigcap B_2 = \{c \mid \exists c_1', c_2',\ c_1' \in CMXS(c_1,t)$, and $c_2' \in CMXS(c_2,t),\ c \preceq c_1' \cap c_2'\}$ = B.

Based on Proposition 5, $A_1 = B_1$, $A_2 = B_2$. Consequently, $A = A_1 \bigcap A_2 = B_1 \bigcap B_2$ = B.

□

Proposition 7.

PROOF. Let A = $\{c \mid c \prec t \;\&\&\; c \not\preceq c_1\}$. B = $\{c \mid c \prec t \;\&\&\; \exists c_1' \in CMNS(c_1,t), c_1' \preceq c\}$. $CMNS(c_1,t)$=$\{(p_{x_i}, v_{x_i}) \mid (p_{x_i}, v_{x_i}) \in t\backslash c_1\}$.

First we will show A ⊆ B.

With respect to set A, $\forall c' \in A$, it has $c' \prec t$ and $c' \not\preceq c_1$. That is, $\forall e \in c', e \in t$, and $\exists e' \in c', e' \notin c_1$. Hence, $\{e'\} \preceq c'$, $e' \in t\backslash c_1$, which indicates that $c' \in \{c \mid c \prec t \;\&\&\; \exists c_1' \in CMNS(c_1,t), c_1' \preceq c\}$ = B, so A ⊆ B.

Second we will show B ⊆ A.

With respect to set B, $\forall c' \in B$, it has $c' \prec t$ and $\exists c_1' \in CMNS(c_1,t), c_1' \preceq c'$. As $c_1' \in CMNS(c_1,t)$, $\exists e' \in t\backslash c_1, c_1' = \{e'\}$. Hence, $\{e'\} \preceq c'$. Hence, $c' \not\preceq c_1$. Consequently, $c' \in \{c \mid c \prec t \;\&\&\; c \not\preceq c_1\}$ = A, which indicates that B ⊆ A.

□

Proposition 8.

PROOF. We just need to prove that for two healthy schemas c_1, c_2, and a failing test case t ($c_1 \prec t$, $c_2 \prec t$), we have $\{c \mid c \prec t \;\&\&\; \forall c_i \in \{c_1, c_2\}, c \not\preceq c_i\} = \{c \mid c \prec t \;\&\&\; \exists c_1' \in CHFS(c_1,t) \bigvee CHFS(c_2,t),\ c_1' \preceq c\}$.

Let $A = \{c \mid c \prec t \;\&\&\; \forall c_i \in \{c_1, c_2\}, c \not\preceq c_i\}$, $A_1 = \{c \mid c \prec t \;\&\&\; c \not\preceq c_1\}$, $A_2 = \{c \mid c \prec t \;\&\&\; c \not\preceq c_2\}$. It is easily to get $A = A_1 \bigcap A_2$.

Let $B = \{c \mid c \prec t \;\&\&\; \exists c_1' \in CHFS(c_1,t) \bigvee CHFS(c_2,t),\ c \preceq c_1'\}$. Here, $CMXS(c_1,t) \bigvee CMXS(c_2,t) = \{c \mid c = c_1' \cup c_2',\ where\ c_1' \in CMXS(c_1,t),\ and\ c_2' \in CMXS(c_2,t)\}$.

Let $B_1 = \{c \mid c \prec t \;\&\&\; \exists c_1' \in CHFS(c_1,t),\ c_1' \preceq c\}$, and $B_2 = \{c \mid c \prec t \;\&\&\; \exists c_2' \in CHFS(c_2,t),\ c_2' \preceq c\}$. $B_1 \bigcap B_2 = \{c \mid c \prec t \;\&\&\; \exists c_1' \in CHFS(c_1,t),\ c_1' \preceq c \;\&\&\; \exists c_2' \in CHFS(c_2,t),\ c_2' \preceq c\}$. Note that, $c_1' \preceq c \;\&\&\; c_2' \preceq c \equiv c_1' \cup c_2' \preceq c$. Hence, $B_1 \bigcap B_2 = \{c \mid c \preceq t \;\&\&\; \exists c_1', c_2',\ c_1' \in CHFS(c_1,t)$, and $c_2' \in CHFS(c_2,t),\ c_1' \cup c_2' \preceq c\}$ = B.

Based on Proposition 7, $A_1 = B_1$, $A_2 = B_2$. Consequently, $A = A_1 \bigcap A_2 = B_1 \bigcap B_2$ = B.

□

Proposition 9.

PROOF. Let A = $\{c \mid \exists c_1' \in CMXS(FSS^{\perp},t),\ c_2' \in CMNS\ (HSS^{\top},t),\ c_2' \preceq c \preceq c_1'\}$.

Let B = $\{c \mid \exists c_1' \in CMXS(FSS,t), c_2' \in CMNS(\ HSS,t),\ c_2' \preceq c \preceq c_1'\ \}$.

Based on Proposition 6 and 8:

A = $\{c \mid c \prec t \;\&\&\; \forall c_i \in FSS^{\perp}, c_i \not\preceq c \;\&\&\; \forall c_i \in HSS^{\top}, c \not\preceq c_i\}$.

B = $\{c \mid c \prec t \;\&\&\; \forall c_i \in FSS, c_i \not\preceq c \;\&\&\; \forall c_i \in HSS, c \not\preceq c_i\}$.

First we will prove B ⊆ A.

As $FSS^{\perp} \subseteq FSS$, $HSS^{\top} \subseteq HSS$, $\{\ c \mid \forall c_i \in FSS, c_i \not\preceq c\} \subseteq \{\ c \mid \forall c_i \in FSS^{\perp}, c_i \not\preceq c\}$ and $\{\ c \mid \forall c_i \in HSS, c \not\preceq c_i\} \subseteq \{\ c \mid \forall c_i \in HSS^{\top}, c \not\preceq c_i\}$. Hence, $\{c \mid c \prec t \;\&\&\; \forall c_i \in FSS, c_i \not\preceq c \;\&\&\; \forall c_i \in HSS, c \not\preceq c_i\} \subseteq \{c \mid c \prec t \;\&\&\; \forall c_i \in FSS^{\perp}, c_i \not\preceq c \;\&\&\; \forall c_i \in HSS^{\top}, c \not\preceq c_i\}$. That is $B \subseteq A$.

Next we will prove A ⊆ B.

Note that $\forall c \prec t$, if $\forall c_i \in FSS^{\perp}, c_i \not\preceq c$, it must have $\forall c_i' \in FSS, c_i' \not\preceq c$. Because if not so, then $\exists c_i' \in FSS, c_i' \preceq c$.

As $\exists c_i \in FSS^{\perp}, c_i' \in FSS, c_i \preceq c_i'$, hence, $c_i \preceq c_i' \preceq c$, which is contradiction.

Similarly, $\forall c \prec t$, if $\forall c_i \in HSS^{\top}, c \not\preceq c_i$, it must have $\forall c_i' \in HSS, c \not\preceq c_i'$. Because if not so, then $\exists c_i' \in HSS, c \preceq c_i'$. As $\exists c_i \in HSS^{\top}, c_i' \in HSS, c_i' \preceq c_i$, hence, $c \preceq c_i' \preceq c_i$, which is contradiction.

Combining them, we can get $\forall c' \in \{c \mid c \prec t \;\&\&\; \forall c_i \in FSS^{\perp}, c_i \not\preceq c \;\&\&\; \forall c_i \in HSS^{\top}, c \not\preceq c_i\}$, $c' \in \{c \mid c \prec t \;\&\&\; \forall c_i \in FSS, c_i \not\preceq c \;\&\&\; \forall c_i \in HSS, c \not\preceq c_i\}$. That is, A $\subseteq$ B.

As we have shown B $\subseteq$ A, and A $\subseteq$ B, so A = B. $\square$