The pending schema of Combinatorial Testing

XINTAO NIU, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHANGHAI NIE, State Key Laboratory for Novel Software Technology, Nanjing University, China
JEFF Y. LEI, Department of Computer Science and Engineering, The University of Texas at Arlington, USA
XIAOYIN WANG, Department of Computer Science, The University of Texas at San Antonio, USA
FEI-CHING KUO, Swinburne University of Technology, Australia

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, i.e., the Minimal Failure-causing Schemas (MFS). Recently some elementary researches are proposed to handle the MFS identification problem. However, we argue that many of them are still incomplete in terms of the existence of schemas that cannot be determined to be faulty or not yet. These cannot-be-determined schemas, i.e., the pending schemas, would be hidden dangers to the software under testing. Hence, it is important to obtain these pending schemas for these incomplete MFS identification approaches. In this paper, we proposed several propositions to formulate the set of pending schemas and give three equivalent formulas to obtain them, based on which we reduce the complexity of obtaining pending schemas from $O(2^n)$ to $O(\tau^{|FSS^{\perp}|+|HSS^{\top}|})$, where n is the number of factors in the software, while $|FSS^{\perp}|$ and $|HSS^{\top}|$ are two relatively small numbers and independent on the number of n. We conduct a series empirical studies on some real software systems with various number of parameters and values. Our results shows that the incompleteness in very common in the covering arrays and MFS identification approaches. We also observed that the third proposed formula is the most efficient when compared others in most cases.

CCS Concepts: • **Software defect analysis → Software testing and debugging**;

Additional Key Words and Phrases: Pending Schema, Minimal Failure-causing Schema, Combinatorial Testing, Software Testing

ACM Reference Format:
Xintao Niu, Changhai Nie, Jeff Y. Lei, Xiaoyin Wang, and Fei-Ching Kuo. 2010. The pending schema of Combinatorial Testing. *ACM Trans. Web* 9, 4, Article 39 (March 2010), 23 pages. https://doi.org/0000001.0000001

Authors' addresses: Xintao Niu, State Key Laboratory for Novel Software Technology, Nanjing University, 163 Xianlin Road, Qixia District, Nanjing, Jiangsu, 210023, China, niuxintao@gmail.com; Changhai Nie, State Key Laboratory for Novel Software Technology, Nanjing University, 163 Xianlin Road, Qixia District, Nanjing, Jiangsu, 210023, China, changhainie@nju.edu.cn; Jeff Y. Lei, Department of Computer Science and Engineering, The University of Texas at Arlington, Arlington, Texas, USA, ylei@cse.uta.edu; Xiaoyin Wang, Department of Computer Science, The University of Texas at San Antonio, San Antonio, Texas, USA, Xiaoyin.Wang@utsa.edu; Fei-Ching Kuo, Swinburne University of Technology, Melbourne, Australia, dkuo@swin.edu.au.

1559-1131/2010/3-ART39 $15.00
https://doi.org/0000001.0000001

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	On	Off	Off	On	PASS
4	On	Off	On	Off	PASS
5	Off	On	Off	Off	Fail

1 INTRODUCTION

The behavior of modern software is 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)[16]. When the number of factors is large, the interactions to be checked increase exponentially, which makes exhaustive testing not feasible. Combinatorial testing (CT) is a promising solution to handle the combinatorial explosion problem [6, 7]. 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, i.e., Minimal Failure-causing schemas(MFS), from all the remaining interactions (schemas) [2, 9].

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 last test case failed. We can get five pair-wise suspicious schemas that may be responsible for this failure. They are respectively (Highlight: Off, Status Bar: On), (Highlight: Off, Bookmarks: Off), (Highlight: Off, Smart tags: Off), (Status Bar: On, Bookmarks: Off), (Status Bar: On, Smart tags: Off), and (Bookmarks: Off, Smart tags: Off). Without additional information, it is difficult to figure out the specific schemas in this suspicious set that caused the failure. In fact, considering that the schemas consist of other number of factors could also be MFS, e.g., (Highlight: Off) and (Highlight: Off, Status Bar: On, Smart tags: Off), the problem becomes more complicated. Generally, to definitely determine the MFS 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 [12] specifically studied the properties of MFS in SUT, based on which additional test cases were generated to identify them. Other approaches to identify the MFS in SUT include building a tree model [18], adaptively generating additional test cases according to the outcome of the last test case [22], ranking suspicious schemas based on some rules [5], and using graphic-based deduction [9], 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 [5, 8, 12–15, 17, 22]or *nonadaptive*–additional test cases are chosen independently and can be executed in parallel [2, 9, 10, 18, 21].

Although many efforts have been devoted to identify the failure-causing schemas from failing test cases, we argue that many of them are still incomplete in terms of the existence of schemas that cannot be determined to be faulty or not yet. Particularly, after identifying the MFS from one failing test case, we wonder that does the schemas other than the identified MFS are guaranteed to be irrelevant to the failure in this failing test case? A related question is that, after identifying the MFS, is there exists any schema in this failing test case that is still cannot be determined to be

faulty or not? To answer these two questions is important, because these cannot-be-determined schemas would be hidden dangers to the SUT. Moreover, we need the measures to evaluate the adequacy of the covering arrays and MFS identification approaches in CT, which is a important key to form the confidence of the developer of the SUT. However, to our best knowledge, no such study has been proposed, especially from a theoretical view.

One simple solution is to exhaustively list all the schemas in one failing test case, and then check them to be faulty or not one by one. However, as we have mentioned before, the complexity of this procedure is 2^n, where n is the number of factors in this test case. Hence, this solution is far from feasible when n is very large.

For all of these, a metric should be proposed to assist in evaluating the completeness of MFS identification approaches, and it should be more efficient than a simple exhaustive testing. In this paper, we proposed the notion of **pending schemas**, which indicates the schemas that cannot be determined to be faulty or not. By calculating the number of pending schemas in one failing test case, we can easily assess the extent to which the MFS identification approaches are incomplete. In fact, by the use of pending schemas, we can also evaluate the incompleteness of traditional covering arrays.

Furthermore, we theoretically analyzed the relationships among schemas by proposing nine novel propositions. Based on them, we gave three equivalent formulas, but with different complexities, to obtain the pending schemas. Among these formulas, Formula 3 helps to reduce the complexity of obtaining pending schemas from $O(2^n)$ to $O(\tau^{|FSS^{\perp}|+|HSS^{\top}|})$, where τ is the number of parameter values in the MFS, and $|FSS^{\perp}|$ and $|HSS^{\top}|$ are two relatively small numbers and independent on the number of parameters n in one test case. Formula 3 is much more efficient at obtaining pending schemas when compared to the exhaustive methods which consecutively checks schemas in one failing test case, especially when n is large.

We conducted a series empirical studies on some real software systems with various number of parameters and values. We first evaluated the incompleteness of traditional covering arrays and different fault localization approaches in CT. We also compared the efficiency of three formulas in terms of obtaining pending schemas. Our results mainly shows that the incompleteness in very common in the covering arrays and MFS identification approaches. We also observed that Formula 3 is the most efficient formula among others in most cases.

Contributions of this paper:

- We showed that the traditional covering arrays and the minimal failure-causing schema model are still incomplete in terms of the determination of schemas to be faulty or healthy.
- We introduced the notion of the pending schema to evaluate the incompleteness of these models in combinatorial testing.
- We proposed several propositions to formulate the set of pending schemas and gave three equivalent formulas to obtain the pending schemas, based on which we reduced the complexity of obtaining pending schemas from $O(2^n)$ to $O(\tau^{|FSS^{\perp}|+|HSS^{\top}|})$, where $|FSS^{\perp}|$ and $|HSS^{\top}|$ are two relatively small numbers and independent on the number of n.
- We conducted a series of experiments to evaluate the incompleteness of traditional covering arrays and MFS identification approaches. Besides, we also evaluated the efficiency of the three formulas on obtaining pending schemas.

The remainder of this paper is organized as follows: Section 2 describes the motivation for this work. Section 3 introduces some preliminary definitions and propositions. Section 4 proposes several important propositions to formally identify the characteristics of the pending schemas. Section 5 evaluates the incompleteness of MFS identification approaches and compares the effectiveness of different approaches for obtaining pending schemas. Section 6 discusses the findings of our research

works. Section 7 summarizes the related works. Section 8 concludes this paper and discusses the future works.

2 MOTIVATION

In this section, we will use several examples to show the incompleteness of traditional covering arrays and well-known MFS identification approaches, respectively. These examples are derived from the MS-Word example listed in the introduction. For simplification, we use integer 0 to represent the state *On* and 1 to represent the state *Off* for each option. For example, the second test case listed in Table 1 can be denoted as (1, 1, 0, 0). Also, we use the intuitive notation (.., v_{n_i} -, v_{n_k}, ..., -) to represent the schemas for the system, where v_{n_i} indicate the value that is assigned to the corresponding factor and '-' indicates that the corresponding factor is not in this schema. For example, (1, 1, -, -) represents the schema (Highlight: Off, Status Bar: Off) in this example. Note that we will introduce a more formal denotation of test case and schema in the following section.

Also, to understand the following examples, we first give two rules. The first rule is that all the schemas in a passing test case are non-faulty, i.e., will not cause failure. The second rule is that any schema contain a MFS is a faulty schema, i.e., will also cause the failure. These two rules are wildly used in the MFS identification approaches [5, 12, 13, 22]. We will discuss the justifications of these two rules later, as well as some issues if these two rules are not hold.

2.1 The incompleteness of covering array

We first consider the traditional covering arrays. To understand the incompleteness of the covering array, we need to check each schema in the test case of the covering array. As we said before, we use the same MS-Word example in the first section. Figure 1 lists the detail of this example. The test cases t_1 to t_5 constitute the covering array shown in Table 1. The complete set of the schemas of each test case is attached at the right side of the corresponding test case. For example, for test case t_1, i.e., (0, 0, 0, 0), all the possible schemas (0, -, -, -), (0, 0, -, -), ..., are listed at the right side of t_1. There are $2^4 - 1 = 15$ schemas in total for each test case in this example.

In this figure, the test case with dark color represents a failing test case, while the test case with white color is a passing test case. The schema with white color is non-faulty, i.e., will not cause a failure, while the schema with dark color is faulty, i.e., any test case contain this schema would fail after testing. At last, the schema with light dark color and dashed outline is the pending schema, indicating that we cannot still determine whether it is faulty or non-faulty.

In this figure, we can first observe that all the schemas in the passing test case is non-faulty. This result is according to the first rule we mentioned before. The second observation is that the schema with the maximal number of factors (4 factors) in a failing test case is a faulty schema. In fact, this schema is the failing test case itself, i.e., (1, 1, 0, 1). This is because the failing test case must contain at least one MFS (otherwise, it will not fail). Hence, the schema which is the test case itself must also contain at least one MFS. As a result, it must be faulty schema according to the second rule we mentioned before. The last observation is that the other schemas in this failing test case t_5 are not guaranteed to be faulty schemas. In fact, if we assume this test case only contains one MFS (1, 1, 0, 1), then all the other schemas can be non-faulty schemas. Hence, these schemas cannot be determined to be faulty or not if we focus on this failing test case alone. As a result, we label these schemas as pending schemas initially.

Combining the three observations, we can further remove some pending schemas in t_5 by selecting the schemas that have already been appeared in the passing test cases. These schemas are (1, -, -, -), (-, -, 0, -), (-, 1, -, -), and (-, -, -, 0), which are labeled as non-faulty schemas. At last, the determination results of these schemas of t_5 can be shown in the "*Status*" row. Note that except for these schemas that have been determined to be faulty and non-faulty, there still exist some pending

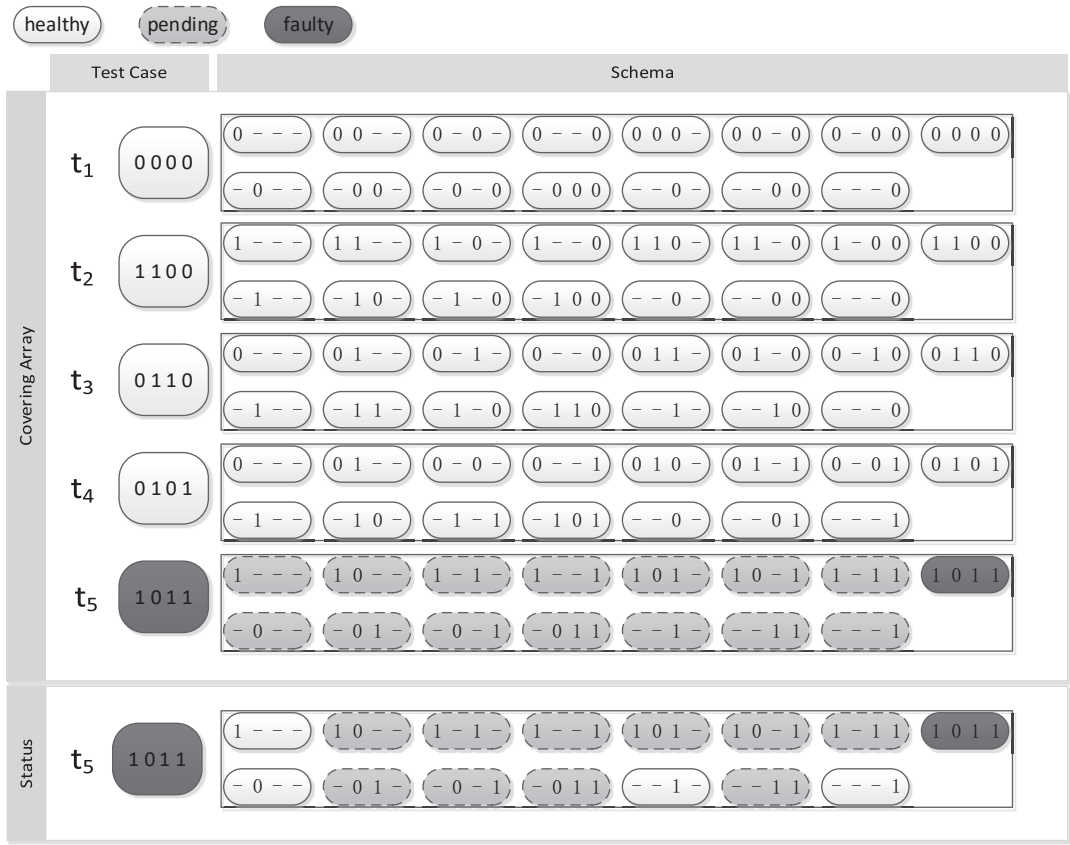

Fig. 1. The incompleteness of Covering array

schemas we cannot further removed by the original two rules. For example, (1, 0, -, -) didn't appear in any passing test case, and it did not contain any identified MFS. Hence, in this example, a single covering array is incomplete because of the existence of these pending schemas.

2.2 The incompleteness of OFOT

Since covering array alone cannot remove all the pending schema in the failing test cases, we need more information to satisfy this target. According to the second rule, i.e., the schema that contain the MFS is faulty schema, one method to reduce the number of pending schemas is to filter out those schemas which contain the MFS. However, without knowing the specific MFS in prior, we can only guarantee that the failing test case itself is faulty schema (it must contain the MFS). In fact, with the covering array alone, this is what we can only do to utilize the second rule.

Hence, to further reduce the set of pending schemas, we need to identify the MFS in the failing test case. One-Factor-One-Time (OFOT) [12] is one of the most wildly used MFS identification approach. It identifies the MFS by modifying the original failing test case to see whether the modification would break the MFS in it. More specifically, at each time, it modifies one factor of the original failing test case and keeps the remaining factors to be as the same as the original failing test case. By doing this, it generates one new test case at each time. It then tests the newly generated test case. If this newly generated test case passes, it indicates the modified factor break

the MFS in the original failing test case, and therefore, the original factor in the failing test case is one factor in the MFS. Otherwise, the original factor in the failing test case is not the factor in the MFS if the newly generated test case fails.

Next, let us use OFOT to identify the MFS and help to narrow down the set of pending schemas of t_5 in the original covering array. First, we assume that there is one MFS (-, -, 1, 1) in failing test case t_5. Then OFOT will work as follows: it generates four additional test cases t_6 to t_9 as shown in Figure 2, respectively, each of which has one factor to be mutated from t_5. Since t_8 and t_9 passed after testing, the original two factors (-, -, 1, -) and (-, -, -, 1) in t_5 are two factors in the MFS. The fails of t_6 and t_7 shows that there is no other factors in this MFS. Hence, OFOT identified the schema (-, -, 1, 1) as the MFS, which is identical to the schema (-, -, 1, 1) that we set as MFS in prior.

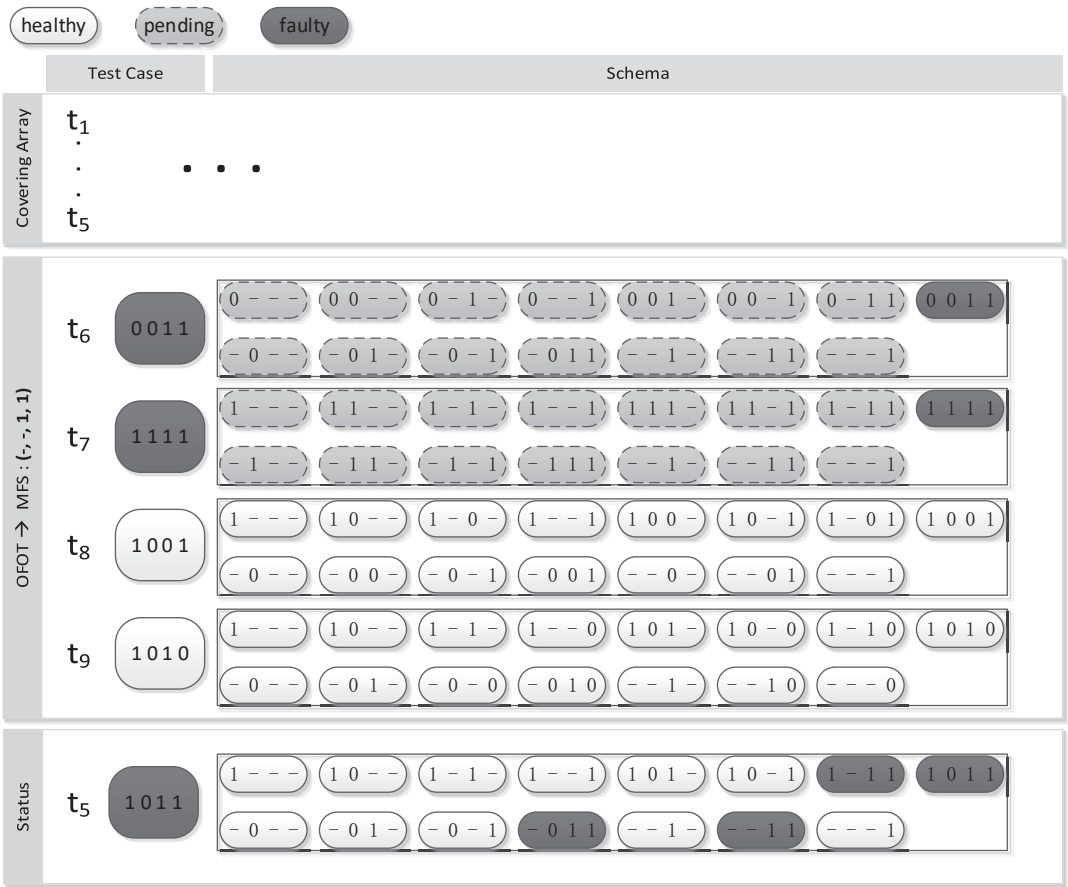

Fig. 2. OFOT with single MFS

To analyse the pending schemas, we first list all the schemas in each additional test case in Figure 2. The same as we observed from the covering array example, for the passing test case t_6 and t_7, all the schemas contained in it are non-faulty. For the failing test cases t_8 and t_9, we initially set the schemas which are failing test cases themselves as faulty schemas. Other schemas in these two failing test cases are all set to be pending schemas initially.

With these additional information, let us re-consider the status of the schemas in the original failing test case t_5. Firstly, as the identified MFS is (-, -, 1, 1), we can remove all the pending schemas of t_5 which contain this schema. These schemas are (-, -, 1, 1), (1, -, 1, 1), (-, 0, 1, 1), and (1, 0, 1, 1), respectively, and are labeled with dark color in the "*Status*" row for t_5 in Figure 2. Next, we remove all the pending schemas of t_5 which appeared in these two passing test case t_6 and t_7. As shown in the "*Status*" row of Figure 2, all the remaining pending schemas are removed and labeled with white color. Hence, we can learn that in this single MFS circumstance, OFOT works perfectly to remove all the pending schemas. However, when the failing test case contains multiple MFS, it does not go that well.

Now let us assume there are two MFS in the failing test case t_5, which are (1, 0, -, -) and (-, -, 1, 1), respectively. At this time, OFOT still generates the same four additional test cases, i.e., t_6 to t_9, as shown in Figure 3. But different from the example of single MFS, all the additional test cases failed at this time. This is because, the strategy of OFOT, i.e., mutating one factor at one time, cannot break all the MFS at the same time. As a result, it cannot identify any of the MFS.

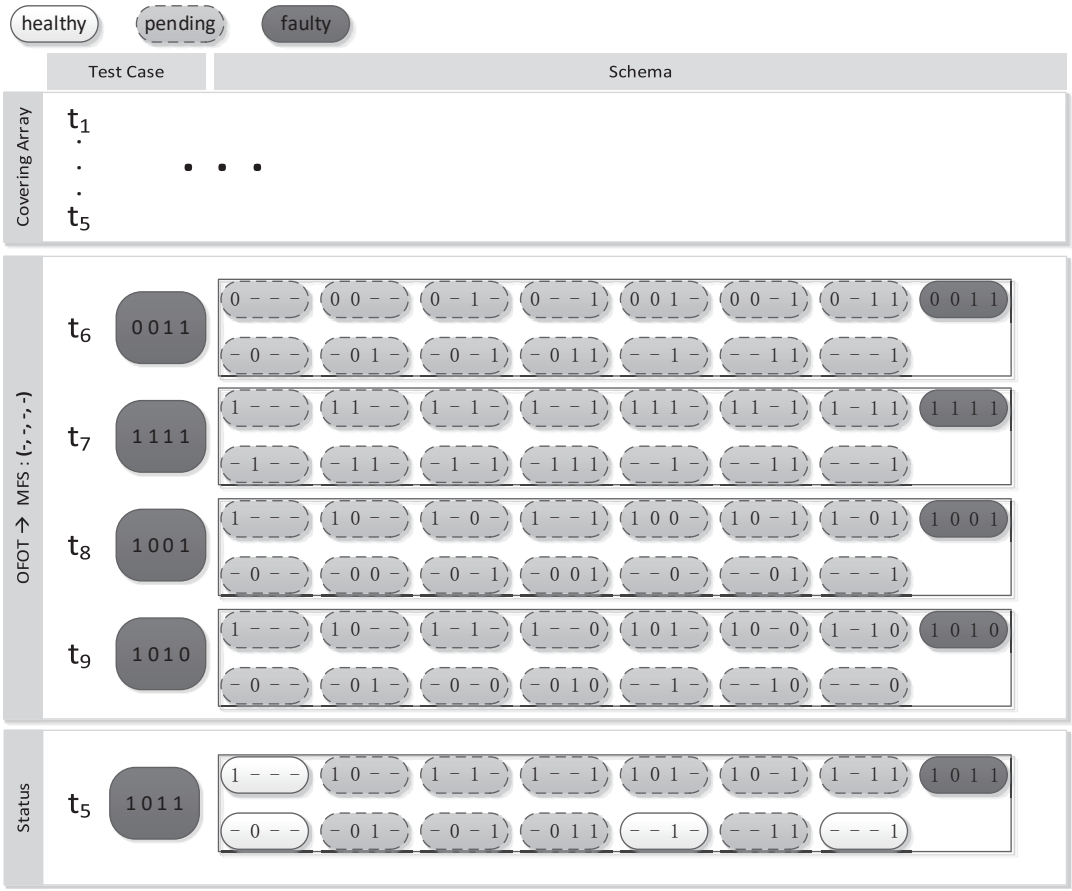

Fig. 3. OFOT with multi MFS

There are two negative influences of this result. First, as we cannot identify the MFS by OFOT, the pending schemas in t_5 that contain the MFS also cannot be determined. Second, as all the test

cases failed after testing, we cannot remove any pending schemas in t_5 that appear in the passing test case. As a result, the status of the pending schemas of t_5 will evolve to the "*Status*" row of Figure 2. We can observe that the status of the schemas of t_5 is the same as the previous example with only using covering array alone. Hence, in the condition that one failing test case contains multiple MFS, the MFS identification approach OFOT is still incomplete.

2.3 The incompleteness of FIC

From the example of OFOT, we can learn the main cause of incompleteness of OFOT is that the failing test case contains multiple MFS. For this, FIC [22] (short for Faulty Interaction Characterization) augmented OFOT to handle the multiple MFS problem. FIC also mutates one factor at a time to generate one additional test case. The only difference is that it will not always rollback to the original value of one factor it has mutated when it goes on mutating other factors (only when a passing test case appears, it will rollback to the original value). This operation will break multiple MFS in one test case and finally there remains only one MFS to identify. We still use the same example of multiple MFS used in OFOT to illustrate how FIC works and to see whether FIC satisfies the completeness criteria. The result is shown in Figure 4.

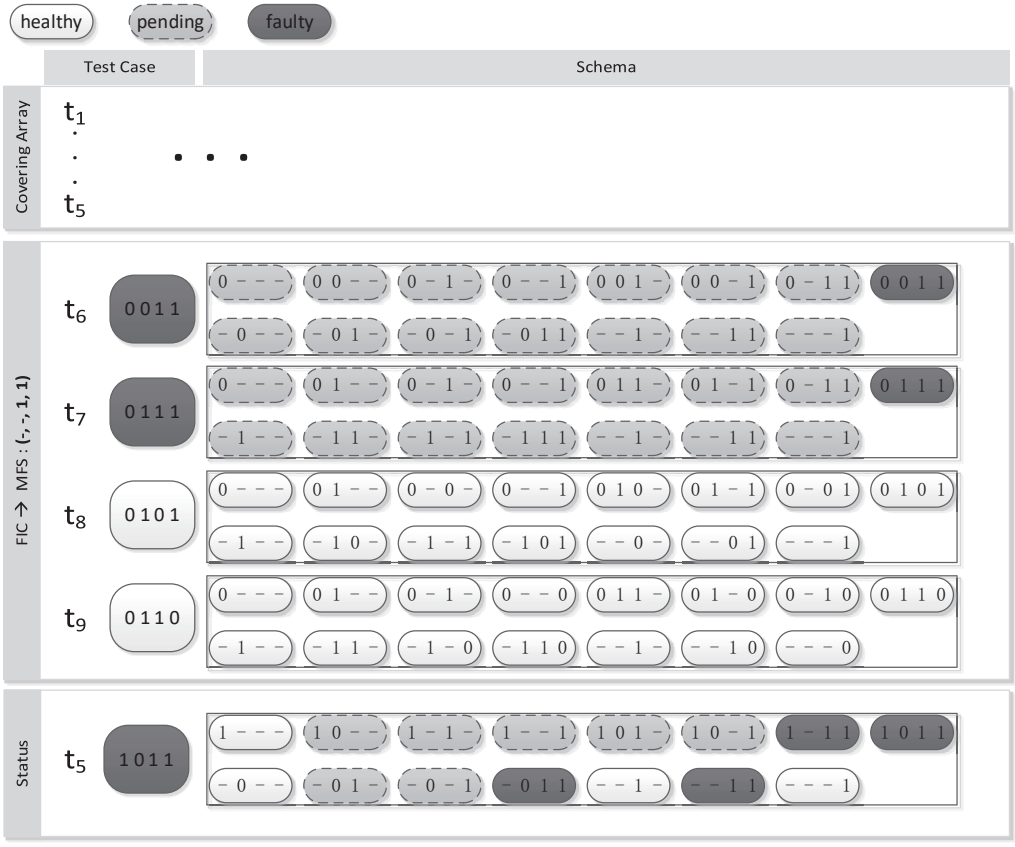

Fig. 4. FIC with multiple MFS

In Figure 4, FIC also generated four additional test cases t_6 to t_9. The first case t_6 is the same as the t_6 generated by OFOT, but the second additional test case t_7 generated by FIC is different from

t_7 generated by OFOT. This is because t_7 generated by FIC keeps the first value of t_6 (which is a failing test case) instead of the original first value of t_5. Note that FIC also keeps this value in the following generated test cases t_8 and t_9. By doing this, FIC forbids the appearance of the MFS (1, 0, -, -) later. The same as the first value of t_6, t_8 and t_9 keep the second value of t_7 because t_7 is also a failing test case. The failings of t_6 and t_7 indicated that there still exists other MFS in the original failing test case t_5, while the first and second factor is not in the remaining MFS.

With respect to the passing test case t_8, test case t_9 did not keep the third value of t_8 but rollback to the value of the original failing test case t_5. This is because, since t_8 passed after testing, there is no MFS in this test case. FIC should rollback this value to keep this factor of the MFS and check if there exists other factor of this MFS or not. After all, the passings of t_8 and t_9 indicated that (-, -, 1, -) and (-, -, -, 1) are two the factors in the remaining MFS. Hence, the MFS identified by FIC is (-, -, 1, 1).

Next, let us check status of the original failing test case t_5. The same as OFOT, we first remove the pending schemas that contain the MFS (-, -, 1, 1), and then remove the pending schemas that appear in the additional passing test case. The result is listed in the "*Status*" row of Figure 4. We can observe that there still exists 6 pending schemas, i.e., (1, 0, -, -), (1, -, 1, -), (1, -, -, 1), (1, 0, 1, -), (1, 0, -, 1), (-, 0, 1, -), and (-, 0, -, 1), respectively.

One reason for this incompleteness that FIC missed anther MFS in the failing test case t_5, which is (1, 0, -, -). The iterative version of FIC [22], i.e., FINOLP is designed to handle this problem. Specifically, after identifying one MFS in the original failing test case, FINOLP first generates one more test case by mutating the factors in the original failing test case which appear in the identified MFS. If the generated test case fails after testing, which indicates that there still exists other MFS, it will use FIC to identify the MFS in this generated test case. This process repeats until the test case which is generated by mutating all the factors in the identified MFS passes. Figure 5 shows the detail when applying FINOLP on this example.

In Figure 5, test cases t_1 to t_9 are the same as those of the example in Figure 4. After identifying the MFS (-, -, 1, 1), FINOLP first generated the test case t_{10} to check whether there exists other MFS in the original failing test case t_5 by mutating the values that appear in this MFS. Since t_{10} failed after testing, it repeated FIC approach on t_{10} to identified the remaining MFS. Therefore, it generated four additional test cases t_{11} to t_{14}. The passings of t_{11} and t_{12} indicated another MFS was (1, 0, -, -). FINOLP continued to check whether there exists any other MFS by generating the test case t_{15}. The passing of t_{15} showed that there did not exist any other MFS in the original failing test case t_5. Above all, FINOLP accurately identified all the MFS we have set in piror.

Next, the same as before, we use the two rules to check the status of the pending schemas in the original failing test case t_5. The result is shown in the "*Status*" row of Figure 5. We can observe that there still exists four schemas, which are (1, -, 1, -), (1, -, -, 1), (-, 0, 1, -), and (-, 0, -, 1), respectively. It is easy to find that these four schemas neither contain any MFS nor appears in any passing test cases. Hence, although FIC and FINOLP can handle the multiple MFS problem, it is still incomplete for the existence of these pending schemas.

2.4 Additional efforts to remove the pending schemas

Since Covering array, OFOT, FIC, and FINOLP cannot remove all the pending schemas, more efforts are needed to accomplish this goal. Note that to clear all the pending schemas is important, because some of them can be potential faulty schemas and may be harmful for this system under testing. For this, we decided to check these remaining pending schemas by generating more test cases that contained them and executing these test cases. If these test cases passes, we can directly use the first rule we mentioned before to determine these schemas to be non-faulty schemas. Otherwise, we need to adopt other methods to determine these schemas to be faulty or not.

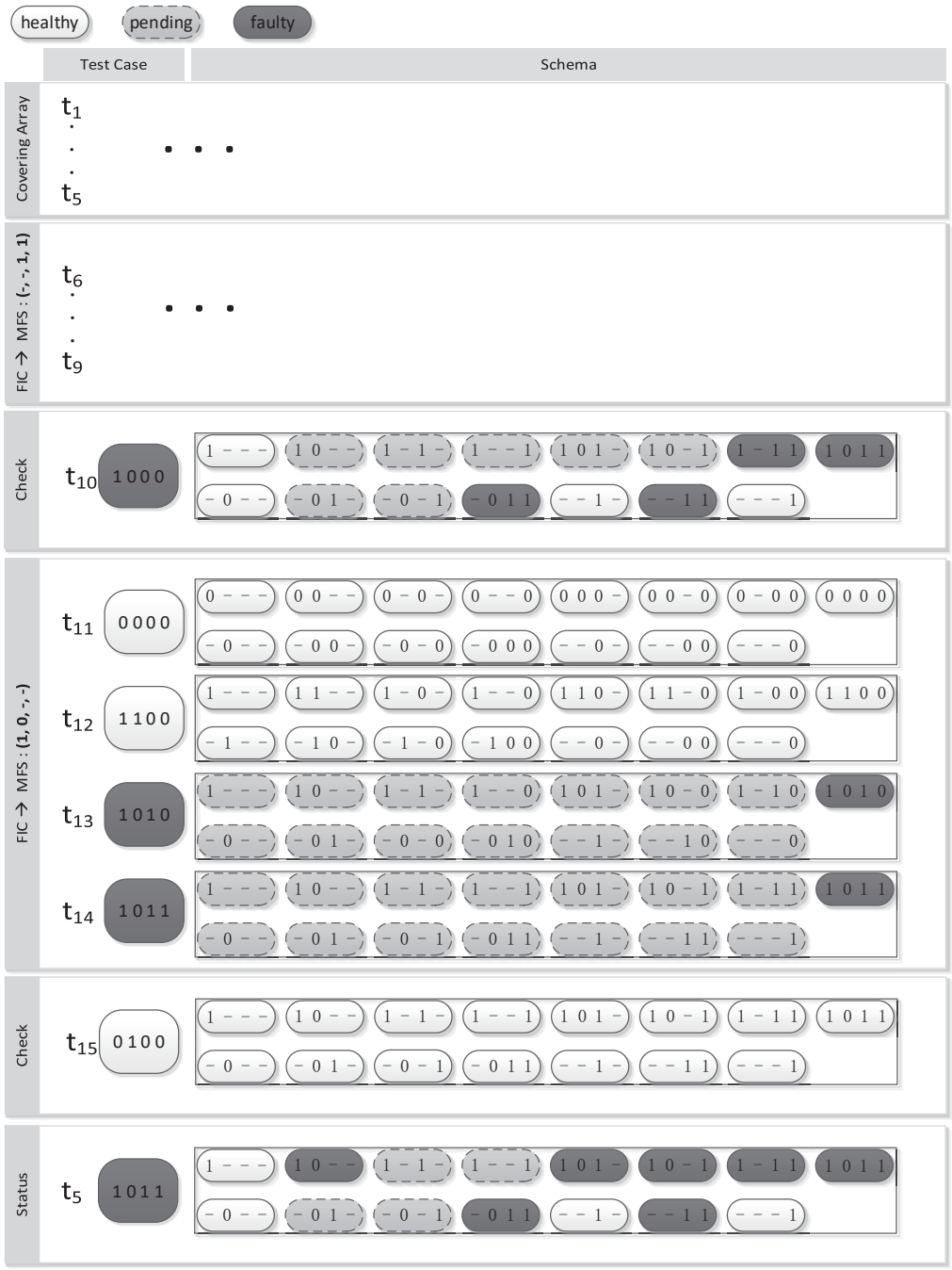

Fig. 5. FINOVLP with multiple MFS

As for this example, we generated four test cases to contain these pending schemas one by one (Note that in this example, we cannot generate one test case contain more than one pending schema

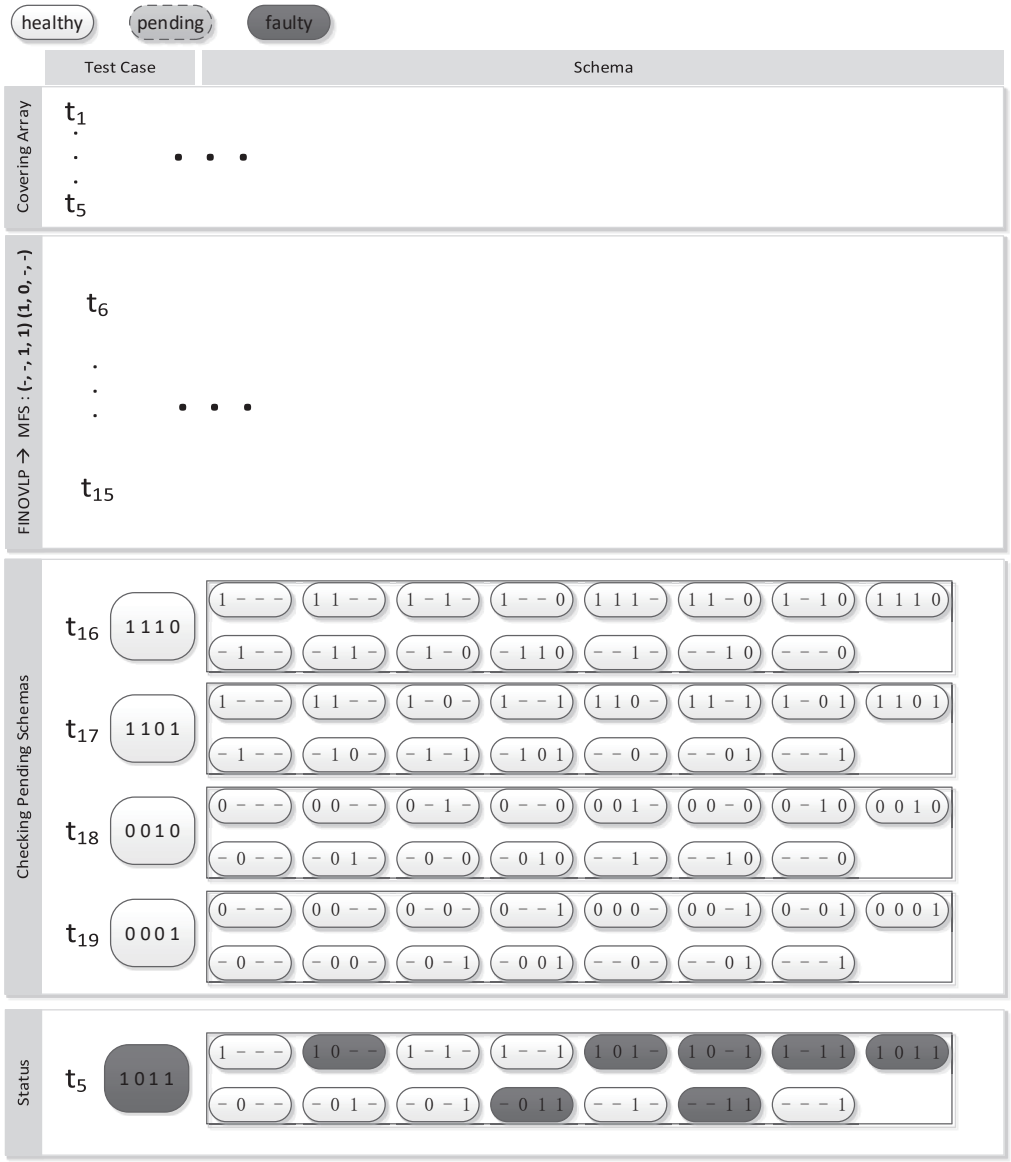

Fig. 6. Additional efforts to remove pending schemas

without including any MFS). The result is shown in Figure 6. In this figure, all the additional test cases, i.e., t_{16} to t_{19}, passed after testing. Hence, all the remaining schemas are non-faulty schemas. The final status of the pending schemas of t_5 is shown in the "*Status*" row of Figure 6. We can observe that all the schemas in t_5 are determined to be non-faulty or faulty. Hence, in this condition, we can guarantee that the test of t_5 is complete.

Note that if any of the test case (t_{16} to t_{19}) failed after testing, we cannot determine whether the corresponding pending schema in that test case is faulty or not. In this case, we need to generate

more test cases to determine the status of this pending schema (If this schema is a faulty schema, the cost will increase exponentially according to the formal definition of faulty schema which will be given later).

2.5 A summary

There are two main observations from this section. First, traditional covering array and MFS identification approaches are still incomplete in terms of the existence of pending schemas which cannot be determined to faulty or non-faulty. Second, to remove all the pending schemas is time-consuming. In fact, just listing all the schemas and checking them one by one is inefficient. In this example, we need to check $2^4 - 15$ schemas for each test case. However, with the increase of the number of factors in one test case, the cost for checking the pending schemas increase exponentially.

For all of these, we need to theoretically analyze the properties of the pending schemas in the failing test cases and to give a more efficient method to obtain them.

3 BACKGROUND

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

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).

Definition 3.1. 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 3.2. For the SUT, the τ-set $\{(p_{x_1}, v_{x_1}), (p_{x_2}, v_{x_2}), ..., (p_{x_\tau}, v_{x_\tau})\}$, 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 this definition of schema is a formal description of schemas we discussed in the section of *Motivation*. For example, the schema (1, -, 0, -) is exactly the schema $\{ (p_1, 1), (p_3, 0) \}$ here. We use this formal definition because it benefits the description of the following theoretical analysis, including these propositions and proofs.

Definition 3.3. 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)}. Also, since a test case itself is a schema, then if a test case t contains a schema s, we have $s \prec t$. For example, the 2-degree schema {(Highlight, Off), (Status Bar, On)} is also a sub-schema of a test case {(Highlight, Off), (Status Bar, On), (Bookmarks, On), (Smart tags, Off)}

Definition 3.4. 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)* [12].

Note that MFS is identical to the failure-inducing interaction mentioned previously. Figuring the MFS helps to identify the root cause of a failure and thus facilitate the debugging process.

Definition 3.5. 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 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 3.6. 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 *cannot-be-determined* schema discussed in Section 1 and Section 2, which is the key schema we focus on in this paper. To analyse the pending schemas in one failing test case is important to evaluate the completeness of the test.

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[5, 13, 22]. 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 [3, 18] 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 [11] may happen through testing, when the faulty-code is executed but the failure doesn't propagate to the output. Masking effect [19] 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 foundations of the theory of pending schemas.

Proposition 3.7 (Transitive). *Given schemas s_1, s_2, and s_3, if $s_1 \prec s_2$, $s_2 \prec s_3$, then $s_1 \prec s_3$.*

This proposition shows the transitivity of the subsuming relationships of schemas, its proof can be directly derived from the definition of the schema.

Proposition 3.8 (Sub of Healthy). *Given a healthy schema s_1, then $\forall s_2, s_2 \prec s_1$, s_2 is a healthy schema.*

Proof. According to the definition of healthy schema, there exists at least one test case that contains s_1 and passes. Let this test case be t. Obviously, $s_1 \prec t$. Then $\forall s_2 \prec s_1$, we have $s_2 \prec s_1 \prec t$ according to Proposition 3.7. Hence, $\forall s_2 \prec s_1$, t contains s_2 and passes. According to the definition of healthy schema, $\forall s_2 \prec s_1$, s_2 is a healthy schema. □

This proposition shows that any subschema of a healthy schema is also a healthy schema. In fact, this proposition can deduce the first rule in Section 2.

Proposition 3.9 (Super of Faulty). *Given a faulty schema s_1, then $\forall s_2, s_1 \prec s_2$, s_2 is a faulty schema.*

Proof. let T_2 be the set of all the test cases that contain s_2, we have $\forall t_2 \in T_2$, $s_2 \prec t_2$. Since $s_1 \prec s_2$, we have $\forall t_2 \in T_2, s_1 \prec s_2 \prec t_2$ according to the Proposition 3.7. Hence, $\forall t_2 \in T_2$, t_2 contains s_1. According to the definition of faulty schema, any test case which contains s_1 would fail. Hence, $\forall t_2 \in T_2$, t_2 is a failing test case. According to the definition of faulty schema. s_2 is a faulty schema. □

This proposition shows that any super-schema of a faulty schema is also a faulty schema. Note that his proposition can deduce the second rule in Section 2.

4 PENDING SCHEMA

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.

4.1 What is pending schema

Through the motivating examples, we can learn that the pending schema is not checked to be healthy and faulty. That is,

However, to emulate all the faulty schemas and all the healthy schemas is very large. By the propostios 3 and 4,

More formally, we can learn that it should not be any super-schemas of existing faulty schemas, and any sub-shemas of existing healthy schemas.

Proposition 4.1 (Sandwich rule). *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.*

4.2 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 3.9 and 3.8, it is easy to find that, the pending schemas set PSS should be the following set

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

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 4.2. 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 set 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 4.3 (one necessary condition). *Given a faulty schema c_1, a failing test case t, where $c_1 \preceq t$, we have $\{c \mid c \prec t \;\&\&\; c_1 \not\preceq c\} = \{c \mid \exists c_1' \in CMXS(c_1, t), c \preceq c_1'\}$.*

This proposition means that the pending schema must be the sub-schema or equal to the schemas in CMXS, otherwise, it is a faulty schema.

In the equation of Proposition 4.3, 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 any schema that is not belong to this set is faulty schema according to Proposition 3.9. 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 4.3 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 $\{c \mid \exists c_1' \in CMXS(c_f, t), c \preceq c_1'\}$.

Table 2. An example of Proposition 4.3

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

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 4.4. 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 4.3, we have:

Proposition 4.5. *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 4.5 extends Proposition 4.3 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 $\{c \mid \exists c'_1 \in CMXS(FSS, t), c \preceq c'_1\}$.

Table 3. An example of Proposition 4.5

Test case t	**{c | c ≺ t && ∀c_i ∈ FSS, c_i ⋠ c}**	**CMXS(FSS, t)**	**{c | ∃c'_1 ∈ CMXS(FSS, t) , c ⪯ 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	{$(p_2, 1)$,$(p_4, 1)$}	{$(p_2, 1)$,$(p_4, 1)$}	{$(p_2, 1)$,$(p_4, 1)$}
{$(p_3, 1)$}	{$(p_1, 1)$}		{$(p_1, 1)$}
{$(p_1, 1)$,$(p_2, 1)$}	{$(p_2, 1)$}		{$(p_2, 1)$}
	{$(p_4, 1)$}		{$(p_4, 1)$}

Next, we give the definition of *CMNS*.

Definition 4.6. 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 set as CMNS(c,t)={{(p_{x_i}, v_{x_i})} | $(p_{x_i}, v_{x_i}) \in t \backslash c$}.

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 4.7 (another necessary condition). *Given a healthy schema c_1, a failing test case t, where $c_1 \prec t$, we have $\{c \mid c \prec t \;\&\&\; c \not\preceq c_1\} = \{c \mid c \prec t \;\&\&\; \exists c'_1 \in CMNS(c_1, t), c'_1 \preceq c\}$.*

This proposition means that the pending schema must be the super-schema or equal to the schemas in CMNS, otherwise, it is a healthy schema.

In the equation of Proposition 4.7, 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 also are the super-schemas of or equal to at least one schema in $CMXS(c_1, t)$. Proposition 4.7 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 $c \mid c \prec t \;\&\&\; \exists c'_1 \in CMXS(c_h, t), c'_1 \preceq c\}$.

Table 4. An example of Proposition 4.7

Test case t	**{c | c ≺ t && c ⋠ c_h}**	**CMNS(c_h, t)**	**{c | c ≺ t && ∃c'_1 ∈ CMNS(c_h, t), c'_1 ⪯ c }**
{(p_1, 1),(p_2, 1),(p_3, 1),(p_4, 1)}	{(p_4, 1)}	{(p_4, 1)}	{((p_4, 1)}
Heathy schema c_h	{(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)}

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 $CMNS(c_1, t) \bigvee CMNS(c_2, t) = \{\{(p_1, v_1), (p_3, v_3)\}\}$. Based on this, we denote $CMNS(HSS, t)$ for a set of faulty schemas.

Definition 4.8. 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 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 4.9. *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 | c ≺ t && ∀c_i ∈ HSS, c ⋠ c_i} = {c | c ≺ t && ∃c'_1 ∈ CMNS(HSS, t), $c'_1 \preceq c$}.*

Similar to Proposition 4.3 and 4.5, Proposition 4.9 extends Proposition 4.7 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 $\{c \mid c \prec t \;\&\&\; \exists c'_1 \in CMNS(HSS, t), c'_1 \preceq c\}$.

Table 5. An example of Proposition 4.9

Test case t	**{c | c ≺ t && ∀c_i ∈ HSS, c ⋠ c_i}**	**CMNS(HSS, t)**	**{c | c ≺ t && ∃c'_1 ∈ 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)}

Based on Proposition 4.5 and 4.9, 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

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

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

As an example, consider a failing test case $t = \{(p_1, 1), (p_2, 1), (p_3, 1), (p_4, 1)\}$, the faulty schema set $FSS = \{\ \{(p_1, 1), (p_2, 1), (p_3, 1)\}, \{(p_1, 1), (p_2, 1)\}\ \}$, and the healthy schema set $HSS = \{\ \{(p_2, 1), (p_3, 1), (p_4, 1)\}, \{(p_2, 1), (p_3, 1)\}\ \}$. It is easy to learn that the minimal faulty schema set $FSS^{\perp} = \{\{(p_1, 1), (p_2, 1)\}\}$, and the maximal healthy schema set $HSS^{\top} = \{\{(p_2, 1), (p_3, 1), (p_4, 1)\}\}$.

Fig 7 lists all the faulty schemas, healthy schemas, and pending schemas of test case t. At the second part, it lists the $CMXS(FSS, t)$, $CMNS(HSS, t)$, and the schema set $\{c \mid \exists c_1' \in CMXS(FSS, t), c_1' \in CMNS(HSS, t), c_2' \preceq c \preceq c_1'\ \}$. At last it shows $CMXS(FSS^{\perp}, t)$, $CMNS(HSS^{\top}, t)$ and $\{c \mid \exists c_1' \in CMXS(FSS^{\perp}, t), c_2' \in CMNS(\ HSS^{\top}, t), c_2' \preceq c \preceq c_1'\ \}$.

We can learn that these two schema set mentioned in Proposition 4.10 are identical in Fig 7, but the schema set based on $CMXS(FSS^{\perp}, t)$, $CMNS(HSS^{\top}, t)$ used much fewer schemas (3 in total) to obtain the result than the schema set based on $CMXS(FSS, t)$, $CMNS(HSS, t)$ (7 in total). Another observation from this figure is that these two schema set are both identical to the pending schemas. It suggests that using CMXS and CMNS can effectively and efficiently to get the pending schemas, when compared to list all the schemas in a test case and find the pending schema one by one according to Formula 1.

At last, according to Proposition 4.10, we have the third formula to compute pending schemas as follow:

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

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

4.3 Getting back to the motivation example

When applying this analysis, we can get back to the motivation example. We do not need list all the schemas, but we can figure out which one is the pending schema (Covering array, OFOT, OFOT with single MFS, FIC, FIC with multiple)

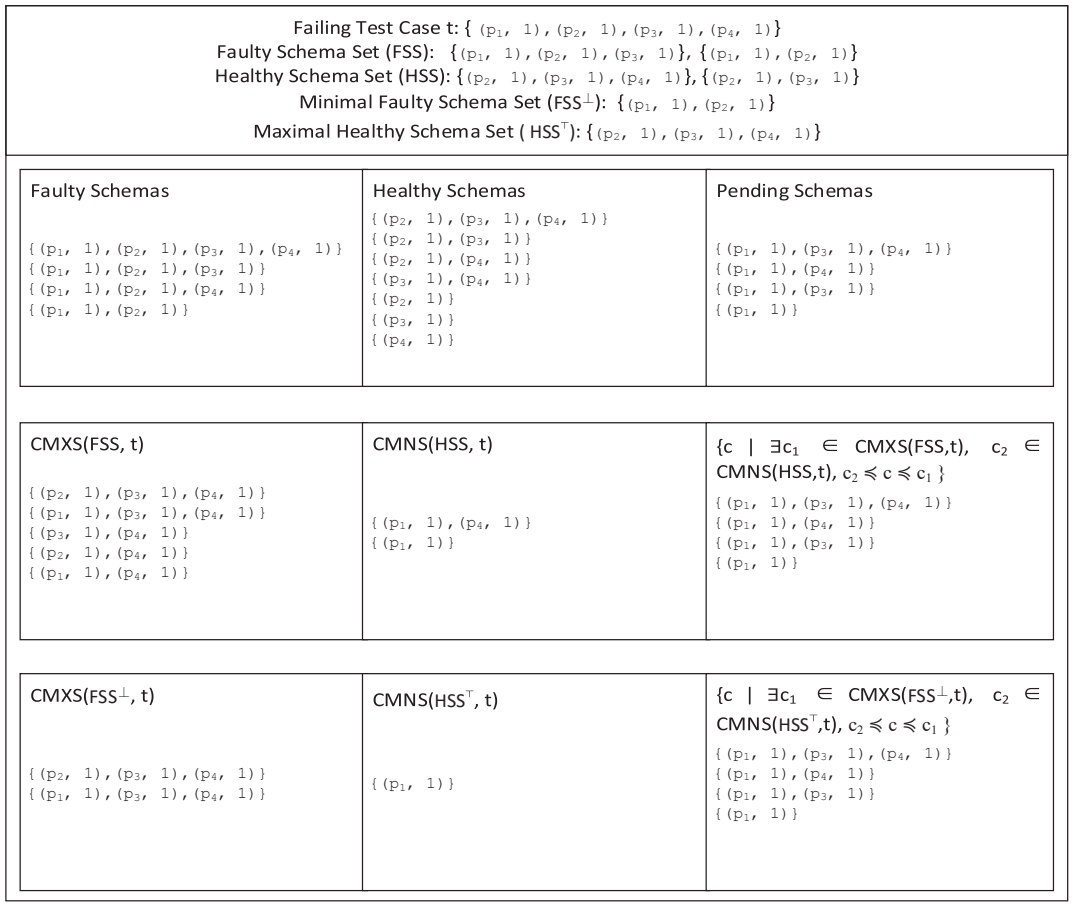

Fig. 7. The example of Proposition 4.10

5 EMPIRICAL STUDIES

5.1 The pending schemas for covering arrays

5.2 The existence of pending schemas for different MFS identification approaches

5.3 The characteristics of pending schemas with various types of MFS (multiple, overlapped, single, low-high degrees)

5.4 The effectiveness of the approach

5.5 Threats to validity

6 DISCUSSION

7 RELATED WORKS

Shi and Nie [15] presented an approach for failure revealing and failure diagnosis in CT , which first tests the SUT with a covering array, then reduces the value schemas contained in the failing test case by eliminating those appearing in the passing test cases. If the failure-causing schema is found in the reduced schema set, failure diagnosis is completed with the identification of the specific input values which caused the failure; otherwise, a further test suite based on SOFOT is

developed for each failing test case, and the schema set is then further reduced, until no more faults are found or the fault is located. Based on this work, Wang [17] proposed an AIFL approach which extended the SOFOT process by adaptively mutating factors in the original failing test cases in each iteration to characterize failure-inducing interactions.

Nie et al. [12] introduced the notion of Minimal Failure-causing Schema(MFS) and proposed the OFOT approach which is an extension of SOFOT that can isolate the MFS in the SUT. This approach mutates one value of that parameter at a time, hence generating a group of additional test cases each time to be executed. Compared with SOFOT, this approach strengthens the validation of the factor under analysis and can also detect the newly imported faulty interactions.

Delta debugging [20] is an adaptive divide-and-conquer approach to locate interaction failure. It is very efficient and has been applied to real software environment. Zhang et al. [22] also proposed a similar approach that can efficiently identify the failure-inducing interactions that have no overlapped part. Later, Li [8] improved the delta-debugging based approach by exploiting useful information in the executed covering array.

Colbourn and McClary [2] proposed a non-adaptive method. Their approach extends a covering array to the locating array to detect and locate interaction failures. Martínez [9, 10] proposed two adaptive algorithms. The first one requires safe value as the assumption and the second one removes this assumption when the number of values of each parameter is equal to 2. Their algorithms focus on identifying faulty tuples that have no more than 2 parameters.

Ghandehari et al. [5] defined the suspiciousness of tuple and suspiciousness of the environment of a tuple. Based on this, they ranked the possible tuples and generated the test configurations. They [4] further utilized the test cases generated from the inducing interaction to locate the fault.

Yilmaz [18] proposed a machine learning method to identify inducing interactions from a combinatorial testing set. They constructed a classification tree to analyze the covering arrays and detect potential faulty interactions. Beside this, Fouché [3] and Shakya [14] made some improvements in identifying failure-inducing interactions based on Yilmaz's work.

Our previous work [13] 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

To identify the MFS in the SUT is important because it provides additional supports to find the cause of the failure. Although many efforts have been proposed to make the MFS identification approaches more efficient and effective, the existence of pending schemas still make them incomplete, which would be hidden dangers to the software under testing. Hence, to identify these pending schemas is of great importance to reveal such potential risks.

In this paper, we studied the characteristics of the relationships among faulty schemas, healthy schemas, minimal failure-causing schemas, and maximal healthy schemas. The subsuming relationships among them form the basis of the methodology for pending schemas. Particulary, we proposed several propositions to formulate the set of pending schemas and give three equivalent formulas to obtain them. Among the three formulas, the last formula reduce the complexity of obtaining pending schemas from $O(2^n)$ to $O(\tau^{|FSS^{\perp}|+|HSS^{\top}|})$, where n is the number of factors in the software, while $|FSS^{\perp}|$ and $|HSS^{\top}|$ are two relatively small numbers and independent on the number of n.

We conduct a series empirical studies on some real software systems with various number of parameters and values. Our results shows that the incompleteness in very common in the covering

arrays and MFS identification approaches. We also observed that the third proposed formula is the most efficient when compared others in most cases.

As a future work, it is appealing to take the advantages of the pending schemas to assist in the MFS identification approaches. More specifically, we would like to develop a new framework for MFS identification, which take the pending schemas as a indicator. That is, any MFS identification algorithm that runs on this framework aims to empty the set of pending schemas. It is of interest to evaluate the efficiency and effectiveness of this framework with respect to various MFS identification approaches that run on it.

A PROOF

We will give the proofs of several important propositions.

Proposition 4.3.

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 4.5.

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, $B_1 \bigcap B_2 = \{c \mid \exists c_1', c_2',\ c_1' \in CMXS(c_1, t), \text{and } c_2' \in CMXS(c_2, t), c \preceq c_1' \cap c_2'\}$ = B.

Based on Proposition 4.3, $A_1 = B_1$, $A_2 = B_2$. Consequently, $A = A_1 \bigcap A_2 = B_1 \bigcap B_2$ = B.

□

Proposition 4.7.

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 $\subseteq$ A.

□

Proposition 4.9.

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 4.7, $A_1 = B_1$, $A_2 = B_2$. Consequently, $A = A_1 \bigcap A_2 = B_1 \bigcap B_2$ = B.

□

Proposition 4.10.

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 4.5 and 4.9:

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 $\subseteq$ 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 $\subseteq$ 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. □

B OTHER EXMAPLES WHEN APPLIED THE PENDING SHEMAS

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their hard work and valuable comments. This work was supported by the National Key Research and Development Plan(No. 2018YFB1003800), National Science Foundation Award CNS-1748109, US Department of Homeland Security(DHS-14-ST-062-001), and US National Institute of Standards and Technologies Award(70NANB15H199).

REFERENCES

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

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

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

[4] Laleh Sh Ghandehari, Yu Lei, David Kung, Raghu Kacker, and Richard Kuhn. 2013. Fault localization based on failure-inducing combinations. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 168–177.

[5] Laleh Shikh Gholamhossein Ghandehari, Yu Lei, Tao Xie, Richard Kuhn, and Raghu Kacker. 2012. Identifying failure-inducing combinations in a combinatorial test set. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 370–379.

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

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

[8] Jie Li, Changhai Nie, and Yu Lei. 2012. Improved Delta Debugging Based on Combinatorial Testing. In *Quality Software (QSIC), 2012 12th International Conference on*. IEEE, 102–105.

[9] Conrado Martínez, Lucia Moura, Daniel Panario, and Brett Stevens. 2008. Algorithms to locate errors using covering arrays. In *LATIN 2008: Theoretical Informatics*. Springer, 504–519.

[10] Conrado Martínez, Lucia Moura, Daniel Panario, and Brett Stevens. 2009. Locating errors using ELAs, covering arrays, and adaptive testing algorithms. *SIAM Journal on Discrete Mathematics* 23, 4 (2009), 1776–1799.

[11] Wes Masri and Rawad Abou Assi. 2014. Prevalence of Coincidental Correctness and Mitigation of Its Impact on Fault Localization. *ACM Trans. Softw. Eng. Methodol.* 23, 1, Article 8 (Feb. 2014), 28 pages.

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

[13] Xintao Niu, Changhai Nie, Yu Lei, and Alvin TS Chan. 2013. Identifying Failure-Inducing Combinations Using Tuple Relationship. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. IEEE, 271–280.

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

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

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

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

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

[19] C. Yilmaz, E. Dumlu, M.B. Cohen, and A. Porter. 2014. Reducing Masking Effects in CombinatorialInteraction Testing: A Feedback DrivenAdaptive Approach. *Software Engineering, IEEE Transactions on* 40, 1 (Jan 2014), 43–66.

[20] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on* 28, 2 (2002), 183–200.

[21] Jian Zhang, Feifei Ma, and Zhiqiang Zhang. 2012. Faulty interaction identification via constraint solving and optimization. In *Theory and Applications of Satisfiability Testing–SAT 2012*. Springer, 186–199.

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

Received February 2007; revised March 2009; accepted June 2009