Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 15, no. 12, December 2019, pp. 3253-3261
DOI: 10.23940/ijpe.19.12.p17.32533261

Mining Minimal Failure-Causing Schema for Software Complex Configuration Space

Liangfen Wei[a], Yong Li[b,c,*], Yong Wang[b,c,d,*], Xiangyu Chen[e], and Zhaohui Xu[f]

[a]*Department of Computer Engineering, Anhui Sanlian University, Hefei, 23060, China*
[b]*Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Ministry of Industry and Information Technology, Nanjing, 210000, China*
[c]*Data Security Key Laboratory, Xinjiang Normal University, Urumqi, 830054, China*
[d]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, 210000, China*
[e]*School of Computer and Information, Anhui Polytechnic University, Wuhu, 241000, China*
[f]*Avic East Optoelectronics (Shanghai) Co., LTD., Shanghai, 201100, China*

Abstract

Minimal failure-causing schema (*MFS*) may be affected by *masking effects* in software complex configuration space. A method for mining *MFS* based on combination testing and its testing results is proposed. The method firstly constructs a combination-fault tree (*CFF-tree*) based on the combined test suite and its results, extracts the frequent parameter-value-combinations from the tree as suspicious *MFS* and calculates their suspiciousness scores, and finally sorts them according to their suspicious scores. Though an iterative framework, *MFS* is repeatedly mined and checked by programmers until a certain stopping criterion is satisfied. Simulation experiments are used to validate the effectiveness of our method with and without *masking effects*. The experimental results show that the proposed method can mine *MFS* in the two scenarios and effectively reduce the number of additional test cases.

Keywords: minimal failure-causing schema; configuration space; CFF-tree; combination testing

(Submitted on September 12, 2019; Revised on October 16, 2019; Accepted on November 25, 2019)

1. Introduction

Combination testing, which involves running software by sampling inputs or triggering software failures in configuration space, is a proven approach for effective software testing with lower costs [1-2]. Research in NIST shows that most system faults are triggered by one or two parameter values, with progressively fewer by three or more. This finding, referred to as the interaction rule, has important implications for software testing because it means that testing parameter combinations can provide more efficient fault detection than conventional methods. The key issue of combination testing is to generate a minimal number of test cases that satisfy certain combination testing coverage criteria. Most combinatorial testing studies focus on generation algorithms to construct effective testing sets and evaluate their effectiveness. In recent years, using the results of combination testing to diagnose the causes of software failures has received great attention. Nie et al. [3] proposed the concept of minimal failure-causing schema (*MFS*) and showed that an *MFS* can help programmers localize and understand fault(s) during software debugging. Generally, a software contains n configuration parameters, each parameter contains m parameter values, and their configuration space is m^n. Mining *MFS* in the complex configuration space and testing results is the key to program debugging.

To mine *MFS*, Nie et al. [4] proposed a one-by-one replacement heuristic method, which replaces the failed combinations of parameter values one by one to reduce the search scope for the *MFS*. Xu et al. [5] proposed a fault interaction characteristic (*FIC*) method. The approach uses a failed test case as a seed to generate an additional test case set, adaptively modifies the additional test case set, performs enhanced testing for the software under test, and repeatedly iterates until finding the *MFS*. Zhang et al. [6] constructed a specific tuple relationship tree (*TRT*) to describe the

* Corresponding author.
E-mail address: liyong@live.com, yongwang@ahpu.edu.cn

relationships of all the specific combinations of parameter values. Using *TRT* can effectively reduce the number of additional test cases generated and obtain a precise *MFS*. However, these methods have the potential assumption that running software by a test case that includes an *MFS* will trigger a software failure. Dumlu et al. [7] pointed out that the masking effects in a real system will run software by the test cases containing *MFS* successful execution occasionally. Therefore, in the real system, even if the *MFS* is covered, the software occurred failure is intermittent. It is difficult for the above methods to obtain a satisfactory result without satisfying the potential assumptions. One solution proposed by researchers is to calculate the suspiciousness ranking of each combination and generate a suspicious ranking list for programmers for further diagnosis. Yilmaz et al. [8-9] used a fault classification tree to analyze a given combination test set to detect potential fault combination patterns in the complex configuration space. Shakya et al. [10] used as many as one factor (*OPOT*) criterion at a time to generate as many failed test cases as possible to reduce the imbalance bias for classification trees. However, the suspicious *MFS* constructed by the fault classification tree strongly depends on the selection of the root node.

Regardless of whether masking effects exist in a system, an *MFS* must exist in the failed test cases according to the failure/fault model [11]. Namely, *MFS* frequently appears in failed test cases. Based on the frequent item mining algorithm in [12], we propose a method for mining *MFS* based on *CFF-tree*. Based on the iterative mining framework, additional failed test cases can be added and re-mined until the *MFS* cannot be mined from the existing test case set.

2. Preliminary

The set of software configuration parameters for a software under test (*SUT*) is defined as $v = \{v_1, \cdots, v_i, \cdots, v_n\}$, where the value comes from a discrete set V_i. An *SUT* may interact with multiple parameter values during software execution to cause possible software failure. Assume that a configuration space of an SUT is shown in Table 1.

Table 1. An example configuration space for an SUT

Hardware (H)	Operating System (O)	Explorer (B)	User (U)
PC	Win2000	Explorer	NewUser
Mac	WinXP	Netscape	OldUser
-	OS9	Firefox	-
-	OS10	Mozilla	-

Definition 1 (*test case set*): A test case set for combination testing can be represented as a set $T = \{(v_1, v_2, \cdots, v_n) \mid v_1 \in V_1, v_2 \in V_2, \cdots, v_n \in V_n\}$.

Definition 2 (*k-value mode*): For a test case set of combination testing, an *n-tuple* $[-, \cdots, v_{l_1}, -, \cdots, v_{l_k}, \cdots]$ is called a *k-value* mode ($n \geq k > 0$). In this tuple, v_{l_i} takes a fixed parameter value, and '-' means that the corresponding parameter can take any value from the parameter value set. When *k = n*, this *k-value mode* is a test case. For example, [-, WinXp, -, NewUser] is a two-valued mode.

Definition 3 (*k-value failure-causing schema*): *k-value failure-causing schema* means that test cases containing this *k-value failure-causing schema* will cause software failure. In contrast, for a set of test cases that includes a certain *k-value mode* if at least one test case is successfully executed, the *k-value mode* is said to be the correct *k-value mode*. A *suspicious k-value failure-causing schema* means that the *k-value mode* is highly correlated with the *root cause failure-causing schema*.

Definition 4 (*sub-mode*): Assume that m_1 and m_2 are k-value modes of the *SUT*. If a parameter value determined in m_1 is a subset of the parameter values obtained from m_2, then m_1 is said to be a *sub-mode* of m_2, and m_2 is a *parent mode* of m_1.

Definition 5 (*k-value MFS*): If the *k-value failure-causing schema* is a *k-value MFS*, all its sub-modes are correct *k-value modes*. The *k-value MFS* is the root cause of software failure.

There exists an assumption for the definition of *k-value MFS*. The assumption is that the software must trigger failures if we run the software by the inputs including a *k-value MFS*. However, this assumption is not always true in practice.

Definition 6 (*masking effect*): The masking effect refers to a test case containing a certain *k-valued MFS* that is accidentally executed successfully by its execution environment. Dumlu et al. [7] pointed out that the masking effect may be caused by the fact that, due to the influence of other parameter value patterns, the software returns early in the execution or unexpectedly exits, which causes the software to run without reaching the *MFS*.

Due to the *masking effect*, running a software inputted by a test case that contains an *MFS* does not necessarily trigger a failure. Assuming [-, WinXp, -, NewUser] is a two-valued *MFS*, all test cases that contain this mode may trigger software failures and may also execute successfully by accident. The goal of combination testing is to build a minimal set of test cases to trigger potential software combination failures. Test case generation algorithms for combination testing are usually constructed based on certain coverage criteria.

There are many coverage criteria for choosing the combination of parameter values. The most obvious choice is to choose all the combinations. However, this is impractical when there are more than three parameters and the number of parameter values is defined.

Definition 7 (*T-wise coverage (TWC)*): Each group of *t-value* modes must be combined, and the t-value mode occurs in the test case set more than one time.

If t is equal to the number of parameters m, then *TWC* is equivalent to all combinations. If we assume that the size of parameter values is the same, test cases that satisfy *TWC* will have at least $((\max_{i=1}^{n} m)^t)$, where n is the number of parameter values, and m is the size of each parameter. *TWC* is expensive in accordance with the number of test cases. When $t = 2$, we call *TWC* pair-wise coverage (*PWC*), and some researchers have suggested that going beyond *PWC* does not help much. There exist many algorithms to generate test cases for *TWC* (generally $t \leq 3$).

Definition 8 (*basic selection criteria*): In a system, a basic parameter value is chosen for each parameter, and a basic selection test case is formed by using the basic parameter value for each parameter. Then, the next test cases are chosen by holding all but one basic parameter value constant and using each non-base choice in each other parameters.

The fixed value is defined as the basic selection parameter value, and the other possible values of the parameter are non-base selection parameter values. A test case that consists of basic selection parameter values for all parameters is called a basic test case. According to the basic selection criteria, a new test case generation method selects non-based selected parameter values for each of the basic selection parameters, and the other basic selection parameter values remain unchanged.

The basic selection of test cases can often be seen as an important test case. In our method, the number of failed test cases needs to be increased. Therefore, each failed test case can be used as a basic selection test case. Based on a basic selection test case, new test cases can be generated with *basic selection criteria*. Each new test case will have only one parameter value different from the failed test case, so the testing is more likely to trigger software failures.

3. Iterative Framework

The goal of our method is to use fewer additional tests and generate fewer suspicious combinations to be checked. *MFS* must be included in the failed tests. Therefore, the most frequent itemset in the failed test cases is most likely to be an *MFS*. Our method constructs a *CFF-tree* to store the relationship between the frequent parameter value combinations in the failed tests. Then, the suspicious *MFS* with frequent *top-k* is given based on the *CFF-tree* and submitted to programmers for further inspection.

Considering the cost of additional testing, the mining *MFS* method can be iterative. The main steps are as follows:

1) Generate test cases based on *TWC*, and run the software to obtain testing results.
2) Run the *MFS*-mining algorithm based on the dataset of failed test cases to obtain the suspicious *MFS*.
3) If the *MFS* is found in the *top-k* of the suspicious *MFS* list (k set to 1 in the following simulation experiments), the iterative framework is stopped; otherwise, proceed to step 4.
4) Generate additional test sets based on *the basic selection criteria*, and randomly select one untested test case for testing. If it is successful, proceed to step 4; otherwise, add the failed test case to the basic test case set and return to step 2.

4. Mining Minimal Failure-Causing Schema

4.1. Combination-Faults Frequent Tree Model

In practice, it is desirable to store the combination of failed test parameter values in a compact tree structure to reduce the number of combinations of parameter values to be checked. If the combination of parameter values occurs infrequently in

the failed test case, these parameter values and their combinations can be reduced.

Definition 9 (*support*): The degree of support of a rule $a \to b$ is expressed as the probability of *a* and *b* occurring simultaneously in the testing set, which is denoted as the co-occurrence probability of *a* and *b*, or $\text{support}(a \to b)$ [13].

$$\text{support}(a \to b) = \text{p}(a \cup b) \tag{1}$$

Definition 10 (*combination-fault frequent tree, CFF-tree*): A *CFF-tree T* can be represented as a $T = \{root, ips, fht\}$, where *root* is the root node, the *root* is usually marked as *Null*, *ips* is a child node set for a prefix subtree, and *fht* is a head pointer table for frequent items.

Definition 11 (*item prefix subtree, ips*): Each tree node of the subtree includes three domains: *item-name*, *count*, and *node-link*, where *item-name* indicates the identity of the item, *count* records the number of sub-paths, and *node-link* links to the next node in the *CFF-tree* carrying the same *item-name* or "*NULL*" if there is none.

Definition 12 (*frequent-item header table, fht*): The *fht* consists of two domains: *item-name* and *head-of-node-link*. *item-name* is the name of the item, and *head-of-node-link* is a node that points to the first *item-name* in the *CFF-tree*.

Based on the definition of the *CFF-tree*, the *CFF-tree* is constructed as shown in Algorithm 1.

Algorithm 1: *CFF-tree* Construction

Input: Γ_f : Failed test suite

Output: *T*: *CFF-tree*

1 Scan Γ_f , frequent itemsets *F* and their frequent item *count*s of the parameter values are collected; the *F* is rearranged in descending order of frequency degrees as *L*, and *L* is a frequent itemset of parameter values.
2 Generate root node *root* for *CFF-tree T*, and labeled as "*Null*";
3 For each *trans* in *L* Do
4 $p \leftarrow getFirstElement(trans)$
5 $P \leftarrow get\,\text{Re}\min derElement(trans)$
$insert\_tree([p \mid P], T)$
/*The function $insert\_tree([p \mid P], T)$ is performed as follows:
6 If *T* has a child *N* such that *N.item-name = p.item-name*, then increment *N*'s count by *1*; else create a new node *N*, and let its count be *1*, its parent link is linked to *T*, and its *node-link* is linked to the nodes with the same *item-name* via the node-link structure. If *P* is nonempty, call *insert_tree(P*; *N)* recursively.*/
7 end for
8 return *T*

4.2. Mining Minimal Failure-Causing Schema

Given a *CFF-tree* and frequent threshold (which can be viewed as *support*), for each frequent parameter value, extracting its frequent itemset consists of the following three steps. The algorithm is shown in Algorithm 2.

Step 1 Obtain the *conditional pattern base* of this frequent parameter value from the *CFF-tree*, which is a path set tailed with the parameter value.

Step 2 Build a *conditional CFF-tree* using the *conditional pattern base*.

Step 3 Repeat Steps 1 and 2 until the *CFF-tree* contains a single path.

Algorithm 2: *Mining-MFS*

Input: *CFF-tree*: Combination-fault frequent trees, α : Combination fault frequent itemsets, starting with $\varnothing$, M_s: Minimum support

Output: *CFPS:* Combined fault frequent itemsets

1 $CFPS \leftarrow \varnothing$
2 if *CFF-tree* contained a single path *P* then
3 for each combination β for a node in *P* do
4 generate itemsets $\alpha \cup \beta$, where *support* greater than or equal to M_s of β
5 end for

```
6        return CFPS ← CFPS ∪ meet with α ∪ β for M_s
7    else // Handle multiple paths
8        for each frequent item α_f of fht in CFF-tree do
9            generate frequent itemset β ← α ∪ α_f , where its support greater than or equal to α_f of M_s
10           construct conditional pattern base B for β , and construct CFF-tree T_β based on B
11           if T_β ≠ ∅ then
12               call CFF-location( T_β, β , M_s)
13           end if
14       end for
15   end if
16   return CFPS
```

4.3. An Illustrating Example

Assume that an *SUT* contains four configuration parameters *a*, *b*, *c*, and *d*, and the parameters are equivalently divided: $a \in \{1,2\}$, $b \in \{1,2,3\}$, $c \in \{1,2\}$, and $d \in \{1,2,3\}$. Based on *TWC*, nine test cases can be generated for *t = 2*. Assuming that the *MFS* is <*b=2*, *c=2*>, the test cases and their results are shown in rows 1 to 9 in Table 2, where test 4 and test 6 are failed.

To add the number of failed test cases, we perform additional testing on failed test cases 4 and 6 using *basic selection criteria*. A total of eight additional test cases are generated. There is a test case that is already in the previous test case set, and it is thus ignored. Therefore, a total of seven test cases are generated. Additional test cases and their testing results are shown in rows 10-16. Due to the influence of *masking effects*, we assume that test case 13, which contains the *MFS*, should have been executed successfully. Successful test case 14, which also contains the *MFS*, can be excluded in the context of the precise *MFS* localization methods. Therefore, with the influence of *masking effects*, it is difficult to use a precise *MFS* localization method to perform program debugging.

Table 2. Combination test case sets and test results

Test Id	a	b	c	d	Test Result
1	2	1	2	1	P
2	1	1	1	2	P
3	2	1	1	3	P
4	1	2	2	1	F
5	2	2	1	2	P
6	1	2	2	3	F
7	2	3	1	1	P
8	1	3	2	2	P
9	2	3	2	3	P
10	2	2	2	2	F
11	1	1	2	1	P
12	1	2	1	1	P
13	1	2	2	2	P
14	2	2	2	3	F
15	1	1	2	3	P
16	1	2	1	3	P

We use an example to illustrate Algorithm 1. Using the transaction database shown in Table 3, the compact data structure can be designed based on the following observations. Given the set of failed test cases in Table 3, use Algorithm 1 to build the *CFF-tree* as shown in Figure 1.

Table 3. Failed test cases and frequent itemsets for the example

TID	Failed test cases	(Ordered) Frequent items
1	a=1, b=2, c=2, d=1	"b2", "c2", "a1"
2	a=1, b=2, c=2, d=3	"b2", "c2", "a1", "d3"
3	a=2, b=2, c=2, d=2	"b2", "c2", "a2"
4	a=2, b=2, c=2, d=3	"b2", "c2", "a2", "d3"

As shown in Figure. 1, the condition base of the parameter value "d3" is {<"b2", "c2", "a1", "d3">:1, <"b2", "c2", "a2", "d3">:1}, and the condition pattern base of the parameter value "a1" is {< "b2", "c2", "a2">:2}.

Obviously, if the parameter value "a1" is a frequent item, the frequent itemset suffixed with the parameter value "a1"

can solve its *condition pattern base*. If the *conditional pattern base* is a single path, for example, the frequent itemset suffixed by "a1" is {"b2", "c2"} combined with "a1", the result is {{"b2", "a1"},{"c2", "a1"},{"b2", "c2", "a1"}}; otherwise, recursively solve its *conditional pattern base* and combine the results. For example, the condition pattern base of the parameter value "d3" is multiple paths {<"b2", "c2", "a1", "d3">:1, <"b2", "c2", "a2", "d3">:1}, and suppose the minimum support degree is 2. Although "a1" and "a2" are frequent items that satisfy a minimum support degree of 2, {"a1", "d3"} and {"a2", "d3"} do not satisfy *support* = 2. Therefore, the condition that *CFF-tree* is constructed by "d3" should be removed from "a1" and "a3". The two paths are merged into one path when the conditional *CFF-tree* is built, but the path is processed. Therefore, the frequent items ending in "d3" with *support=2* are {{"d3"}, {"b2", "d3"}, {"c2", "d3"}, {"b2", "c2", "d3"}}.

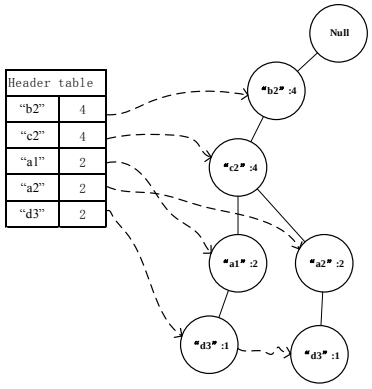

Figure 1. CFF-tree for the example

However, the number of suspicious *MFS* in the final frequent itemset may still be very large. We further reduce the number of suspicious *MFS*. The reduction strategy is as follows: for the two frequent itemsets {{"d3":2}} and {{ "a2", "d3"}:2}, because the latter contains the former and their frequency is the same, the combination of the two frequent parameter values can be merged into {{"a2", "d3" }:2}.

Finally, we give *top k* (*k* can be set by the programmers) of the frequent items as the suspicious *MFS* and submit it to the programmers to be manually checked.

5. Simulation Experiment

We conducted simulation experiments to verify the effectiveness of our approach. The simulation experiment mainly answers the following questions:

RQ1: How effective is our approach for SUTs without *masking Effects*?
RQ2: How effective is our approach for SUTs with *masking Effects*?

5.1. Answer RQ1

For research question 1, we assume that the *SUT* contains eight, ten, and 12 parameters, respectively, and there are three parameter values for each parameter. The test cases are generated using the *IPOG* algorithm recommended by the combination testing tool *ACTS* [14]. For example, the *SUT*'s configuration contains eight parameters, 15 test cases are generated for $t = 2$, 60 test cases are generated for $t = 3$, and 191 test cases are generated for $t = 4$.

To evaluate the effectiveness of our approach for an *SUT* without masking effects, the *CFF-tree* method is evaluated using the tuple relationship tree model (*TRT*) proposed in [7] as a benchmark. The method proposes a relational tree model to identify the *MFS*. Specifically, the method first constructs a relational tree to record all the tuples to be measured and the relationship between them, and then identifies *MFS* based on the tuple relationship. The *TRT* method gives four test case selection strategies, and we use its optimal test case selection strategy (abbreviated *path method*) as a comparison standard. Different from the *CFF-tree* method, the *TRT* method generates a precise *MFS*. According to the definition of *MFS*, it is necessary to check whether the sub-tuples of the *MFS* are correct tuples. For a system with *masking effects*, the method is not applicable.

We assume there is an *MFS* for an SUT without the influence of *masking effects* and set the test case containing the

MFS as a failed test case. We divide test case sets into successful test case sets and failed test case sets. In the process of *MFS-mining*, the iterative framework is used to generate additional test cases. For the three different configuration spaces, we experiment 100 times using different *MFS*s and record the number of additional test cases generated in the *MFS-mining* process. The experiment results are shown in Figure 2. The *CFF-tree* method is superior to the *TRT* method when comparing the number of additional test cases. It is worth mentioning that the *TRT* method uses a precise *MFS-mining* method and needs to check all sub-modes. The number of additional test cases will increase in the process. In theory, the *TRT* method is effective. However, if there exist *masking effects*, this method can hardly achieve ideal results. Different from the *TRT* method, we give the most suspicious *MFS* to programmers to further check. In the experiment, we give the frequent items of the highest frequency *top* 1 as the suspicious *MFS* and submit it to programmers for manual checking. During the testing process, we find that the results of *CFF-tree* are highly correlated with the number of failed test cases. For example, only two or three failed test cases are required for two-way *MFS-mining* and obtain satisfactory results. This also indicates a direction for our future research.

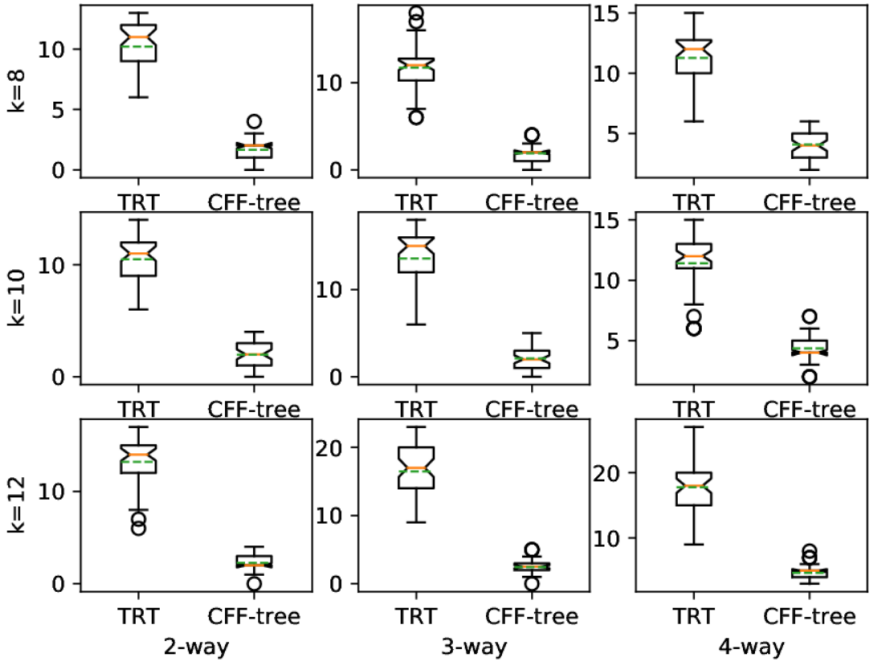

Figure 2. Comparison between *TRT* and *CFF-tree* method

5.2. Answer RQ2

For research question 2, we mainly focus on the effectiveness of our approach for the SUT with *masking effects*, and the different probabilities of the occurrence of *masking effects* in the testing process. We assume that the SUT configuration contains ten parameters. In the SUT with masking effects, the test cases containing the *MFS* are randomly executed successfully with a percentage of 10% and 30%. In the same experiment setting, we compare the number of additional test cases for SUTs with and without *masking effects*.

We use the *ACTS tool* to generate test cases for the SUT configuration based on *TWC* with $t = 2$, $t = 3$, and $t = 4$. Simulate *MFS* with a single *t-value* in the presence of masking effects. Supposing the probability of masking effects is 30%, the test case containing an *MFS* would be executed successfully with a chance of 30%. For each test case that contains an *MFS*, we call a random function that sets the test case as a successful test case with a probability of 30% and set the test case as a failed test case with a probability of 70%.

Similarly, we use our approach to obtain the number of additional test cases for each experiment and repeat 100 times. In each experiment, we use different *MFS*, set the testing results, and record the number of additional test cases for each experiment. The results are shown in Figure 3. In Figure 3, the y-axis represents the number of additional tests. It can be seen that the number of additional test cases increased significantly with the increased probability of masking effects. In the experimental process, we found that masking effects existed in those SUTs, and their testing results are still highly dependent on the number of failed test cases. Therefore, the *CFF-tree* method performs better if there exists a certain

number of failed test cases.

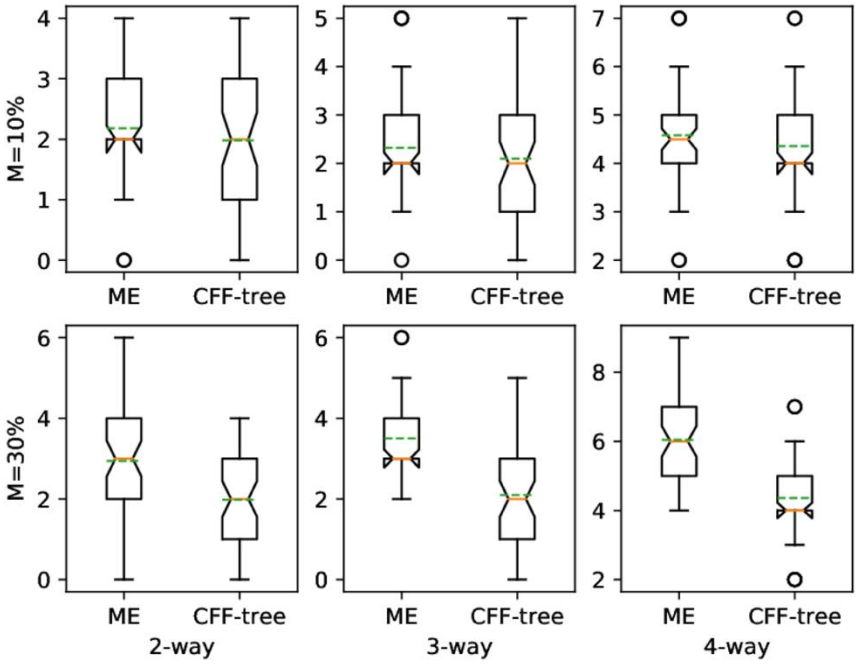

Figure 3. Comparison between existing masking effects (ME) and CFF-tree

6. Conclusions and Future Works

This paper proposes an approach that mines *MFS* for software complex configuration space based on *CFF-tree*. Based on a *CFF-tree* inputted by the failed test case set, we mine the suspicious *MFS* set, and the suspicious *MFSs* are sorted by frequent degree (formally as S*upport)* and submitted to programmers for further checking. This procedure can be repeated by an iterative framework until a stop criterion is satisfied. Preliminary results show that our approach is effective. In particular, our approach has good robustness for an SUT with masking effects.

However, there is a *multi-MFS* scenario in practice [15], and we only consider single-MFS scenarios in our simulation experiments. In future work, it is necessary to further consider the *multi-MFS* scenario and perform experiments to validate the effectiveness of our method. In the case of a *multi-MFS* scenario, the effectiveness of our method is also related to the setting of the *Support* value. Thus, we also need to consider how to set reasonable *Support* values. In addition, we will further validate the effectiveness of our method in actual systems.

Acknowledgments

This work was supported by the Anhui Natural Science Foundation (No. 1908085MF183, 1608085MF147), Anhui University Natural Science Fund Key Project (No. KJ2018A0116, KJ2016A252, KJ2017A104), National Key Research and Development Program (No. 2016YFB100), NSFC of China (No. 61976005, 61772270, 61562087), State Key Laboratory Research Program for Novel Software Technology (Nanjing University) (No. KFKT2019B23), and Safety-Critical Software Key Laboratory Research Program (No. NJ2018014).

References

1. C. Nie and H. Leung, "A Survey of Combinatorial Testing," *ACM Computing Surveys* (*CSUR*), Vol. 43, No. 2, pp. 11, 2011
2. R. Kuhn, R. Kacker, and Y. Lei, "Combinatorial Software Testing," *Computer*, Vol. 42, No. 8, pp. 94-96, 2009
3. L. S. G. Ghandehari, Y. Lei, T. Xie, D. R. Kuhn, and R. Kacker, "Identifying Failure-Inducing Combinations in a Combinatorial Test Set," in *Proceedings of IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 370-379, Montreal, Canada, April 2012
4. C. Nie and H. Leung, "The Minimal Failure-Causing Schema of Combinatorial Testing," *ACM Transactions on Software Engineering and Methodology* (*TOSEM*), Vol. 20, No. 4, pp. 15, 2011
5. B. W. Xu, C. Nie, L. Shi, and H. -W. Chen, "A Software Failure Debugging Method based on Combinatorial Design Approach for Testing," *Chinese Journal of Computers-Chinese Edition*, Vol. 29, No. 1, pp. 132, 2006
6. 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, pp. 331-341, Toronto, Canada, July 2011
7. X. Niu, C. Nie, Y. Lei, and A. Chan, "Identifying Failure-Inducing Combinations using Tuple Relationship," in *Proceedings of 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pp. 271-280, Luxembourg, Luxembourg, March 2013
8. C. Yilmaz, S. Fouche, M. B. Cohen, et al., "Moving Forward with Combinatorial Interaction Testing," *Computer*, Vol. 47, No. 2, pp. 37-45, 2013
9. C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering Arrays for Efficient Fault Characterization in Complex Configuration Spaces," *IEEE Transactions on Software Engineering*, Vol. 32, No. 1, pp. 20-34, 2006
10. E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter, "Feedback Driven Adaptive Combinatorial Testing," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 243-253, Toronto, Canada, July 2011
11. K. Shakya, T. Xie, N. Li, Y. Lei, R. Kacker, and D. R. Kuhn, "Isolating Failure-Inducing Combinations in Combinatorial Testing using Test Augmentation and Classification," in *Proceedings of 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 620-623, Montreal, Canada, April 2012
12. P. Ammann and J. Offutt, "Introduction to Software Testing," Cambridge University Press, 2016
13. J. Han, J. Pei, and M. Kamber, "Data Mining: Concepts and Techniques," Elsevier, 2011
14. L. S. Ghandehari, J. Chandrasekaran, Y. Lei, and L. Sh, "BEN: A Combinatorial Testing-based Fault Localization Tool," in *Proceedings of 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops* (*ICSTW*), pp. 1-4, Graz, Austria, April 2015
15. X. Niu, C. Nie, Y. Lei, K. N. L. Hareton, and X. Wang, "Identifying Failure-Causing Schemas in the Presence of Multiple Faults," *IEEE Transactions on Software Engineering*, June 2018