Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2016, Article ID 2409521, 10 pages
http://dx.doi.org/10.1155/2016/2409521

Research Article

Locating Minimal Fault Interaction in Combinatorial Testing

Wei Zheng, Xiaoxue Wu, Desheng Hu, and Qihai Zhu

College of Software & Micro-Electronics, Northwestern Polytechnical University, Xi'an 710072, China

Correspondence should be addressed to Wei Zheng; zhengweizr@gmail.com

Received 14 October 2015; Revised 22 March 2016; Accepted 19 April 2016

Academic Editor: Henry Muccini

Combinatorial testing (CT) technique could significantly reduce testing cost and increase software system quality. By using the test suite generated by CT as input to conduct black-box testing towards a system, we are able to detect interactions that trigger the system's faults. Given a test case, there may be only part of all its parameters relevant to the defects in system and the interaction constructed by those partial parameters is key factor of triggering fault. If we can locate those parameters accurately, this will facilitate the software diagnosing and testing process. This paper proposes a novel algorithm named complete Fault Interaction Location (comFIL) to locate those interactions that cause system's failures and meanwhile obtains the minimal set of target interactions in test suite produced by CT. By applying this method, testers can analyze and locate the factors relevant to defects of system more precisely, thus making the process of software testing and debugging easier and more efficient. The results of our empirical study indicate that comFIL performs better compared with known fault location techniques in combinatorial testing because of its improved effectiveness and precision.

1. Introduction

Combinatorial testing could significantly reduce test cost and increase quality of software system [1]. It has been proved to be effective especially in a software system where faults come from the interactions of its parameters [2]. Combinatorial testing could detect the parameter interactions that trigger the system faults rather than localize it. If a test case triggers the fault of a system, it reflects that there exists one or more defects in the program [3–5]. However, not all parameters in the test case are relevant to defects. If we are able to locate a parameter in the test case that is relevant to the fault, we can apply this useful information to facilitate the debugging process.

In combinatorial testing, the study on fault interaction technique could be categorized into adaptive method and nonadaptive method according to the dependence between additional test cases and running results [6, 7]. For nonadaptive methods, the generation of additional test cases does not rely on the running result of original test cases. Colbourn and McClary [8] present a nonadaptive method named Locating and Detecting Arrays (LDA). Based on basic known information such as parameters' number, values, and faults' number, the method applies t-way Locating and Detecting Array to locate faults in software system. Martínez et al. [9] present a self-adaptive algorithm based on Errors Locating Arrays (ELA) and analyze the algorithm complexity. However, this method could only be used under the condition that the value's number of each parameter in software is not larger than 2. On the basis of LDA and ELA, Hagar et al. [1] propose the method of Partial Covering Array (PCA) which could be used in the software with known safe value, and it presents a new combinatorial structure to generate ELA.

Another category is known as adaptive method [10], whose generation of additional test cases depends on the information given by the execution of original test cases. Zeller and Hildebrandt [11] present a typical adaptive method named Delta Debugging. The main idea of this method is to identify the interaction that is relevant to the faults by modifying the input parameters. For a test case that triggers the fault, modify some of its input parameters; if the modified test case still triggers the fault, then the modified parameters are irrelevant to fault; otherwise, the modified parameters are related to fault. Based on Delta Debugging, Z. Zhang and J. Zhang [12] present a method named FIC. Similar to Delta Debugging, FIC modify one parameter in a test case

Table 1: The input model of a simple system.

Client browser	Server OS	Client OS	Server database
IE	CentOS	Windows	MySQL
Chrome	Unix	MacOS	Oracle
Netscape	OS/360	Fedora	DB2

with n parameters once. Then repeat this process n times and the minimal fault interaction is calculated afterwards. A restraint of Delta Debugging based methods is that they could only be applicable to the test case containing one minimal fault interaction, but, in real-world programs, more than one minimal fault interaction is very common. Ghandehari et al. [13] present a fault localization tool based on a failure-inducing combinations algorithm of [14]; it leverages the notion of inducing combination to locate the faults inside the source code.

To locate each interaction related to faults, in this paper we present a new complete Fault Interaction Location (comFIL) method. The method includes 2 steps to locate minimal fault interactions. First, after the execution of original test cases, the test cases will be divided into 2 sets: FTS and PTS; the former contains test cases that trigger faults, while the latter contains test cases that do not trigger faults. Then we identify the set of interactions covered by FTS but not covered by PTS and name this set as candidate faulty interaction set (canFIS). Second, we generate additional test cases to select interactions in canFIS and then the minimal fault interaction set is obtained finally.

2. Preliminaries

2.1. Basic Definitions. Assume there is a system under test (SUT) with n input parameters as $P = \{P_1, P_2, P_3, \ldots, P_n\}$. For any $i \in [1, n]$, the range of parameter P_i is denoted as D_i. And pair $\langle P_i, v\rangle$ is used to indicate that the value of parameter P_i is v, also denoted as $(P_i.v)$.

For example, a model named BCSD is shown in Table 1. The model has 4 input parameters: client browser, client operating system (client OS), server operating system (server OS), and server database.

In this model, the range of parameter P_2 is $D_2 = \{\text{CentOS, Unix, OS/360}\}$. The input parameter model of SUT is denoted as $\text{SUT}(n : (|D_1|, |D_2|, |D_3|, \ldots, |D_n|))$, which means SUT has n input parameters and, for each parameter $P_1, P_2, P_3, \ldots, P_n$, the value numbers are $|D_1|, |D_2|, |D_3|, \ldots, |D_n|$, respectively.

Definition 1 (t-way interaction). For a set of t elements $I = \{p_{i_1}.v_1, p_{i_2}.v_2, p_{i_{31}}.v_3, \ldots, p_{i_{n1}}.v_n\}$, if for any m, s, t, $1 \le m$, $s \le t \le n$, there are $i_s, i_m \in \{1, 2, 3, \ldots, n\}$; $(m \ne s,\ i_m \ne i_s)$, then we call I a t-way interaction.

Definition 2 (superinteraction and subinteraction). For 2 interactions, I_1 and I_2, if $I_1 \subset I_2$, then we call I_1 the subinteraction of I_2 and I_2 the superinteraction of I_1. An interaction is both the subinteraction and superinteraction of itself. For example, in model BSCD, interaction $\{P_1.\text{IE}, P_3.\text{MacOS}\}$ is the subinteraction of interaction $\{P_1.\text{IE}, P_2.\text{CentOS}, P_3.\text{MacOS}\}$.

Definition 3 (subinteraction set). For a t-way interaction I, its power set is $\rho(I)$; $\text{subSet}(I) = \rho(I) - \{\Phi\}$ is denoted as the set of all subinteractions of I.

Definition 4 (fault interaction). For an interaction I, if $\forall t \in \{t \mid t \in T_{\text{all}}, I \subset T\}$, $R(t) = \text{fail}$, then I is a fault interaction.

Definition 5 (minimal fault interaction). For an interaction I, if for any interaction $i \in \text{subSet}(I) - \{I\}$, i is not fault interaction, then I is minimal fault interaction.

Definition 6 (additional test case function). For an interaction I, additional test case function generates a test case T that satisfies the following: if $R(T) = \text{fail}$, then I is a fault interaction; otherwise, I is not a fault interaction, denoted as $\text{addTF} : \{\text{ALL\_INTERACTIONS\_IN\_SYSTEM}\} \to T_{\text{all}}$.

Definition 7 (candidate fault interaction set). Denote $\text{canFIS}(T) = \bigcup_{T \in \text{FTS}} \text{subSet}(T) - \bigcup_{T \in \text{PTS}} \text{subSet}(T)$ as a candidate fault interaction set of CA.

2.2. Basic Assumptions

Assumption 1. The parameters in the system are independent of each other.

In many systems, there are constraint conditions between parameters. In [15] Cohen et al. study the method to generate test cases under constraint conditions. However, the constraint conditions among parameters are not the focus of fault location method, so we do not consider the dependence between parameters in this paper.

Assumption 2. If an interaction causes the fault of the system, the test cases containing this interaction will trigger the fault necessarily.

Assumption 3. The test cases generated by additional test case function addTF do not contain the minimal fault interaction.

For an interaction I, addTF generate a test case T; if T trigger the fault, then this fault is caused by interactions in the set of subinteractions of I. If T contains the minimal fault interaction, the complexity of fault location will be increased. So we avoid considering it in this paper. This assumption will be explained later in the implementation of addTF.

The algorithm comFIL presented by this paper is based on the assumptions above. The algorithm is effective only under the condition that all these 3 assumptions are tenable.

2.3. Basic Inferences

Inference 1. The test case that contains fault interaction will trigger the fault of system necessarily and the test case that does not trigger the fault does not contain the fault interaction.

Inference 2. If an interaction is not a fault interaction, any of its subinteractions is not fault interaction.

2.4. Basic Theorems

Theorem 1. *For 2 interactions I_1 and I_2, if $I_1 \subset I_2$, then $subSet(I_1) \subset subSet(I_2)$.*

Proof. From Definition 3 (subinteraction set), we can know that $i \neq \Phi$ and for any $i \in \text{subSet}(I_1)$, $i \subset I_1$, so $i \subset I_2$; namely, $\text{subSet}(I_1) \subset \text{subSet}(I_2)$. □

Theorem 2. *Divide $CA(m, t; (P_1, P_2, P_3, \ldots, P_n))$ into 2 sets: FTS and PTS. In FTS, all test cases triggered the fault of the system, $PTS = CA(m, t; (P_1, P_2, P_3, \ldots, P_n)) - FTS$; if I is the minimal fault interaction of CA, then $I \subset canFIS$.*

Proof. From Inference 1 we can know that, for any minimal fault interaction i in $CA(m, t; (P_1, P_2, P_3, \ldots, P_n))$, there is a failed test case T that satisfies $i \in \text{sunSet}(T)$. So $i \in \bigcup_{T \in \text{FTS}} \text{subSet}(T)$ and $i \in \bigcup_{T \in \text{PTS}} \text{subSet}(T)$; that is, $i \in \text{canFIS}$. □

Theorem 3. *Use MFIS which denotes all minimal fault interactions set of $CA(m, t; (P_1, P_2, \ldots, P_n))$; for a minimal fault interaction i, if $i \in canFIS$ and $R(addTF(i)) = pass$, then $MFIS \cap \bigcup_{I \in i} subSet(I) = \emptyset$ and $MFIS \subset canFIS - \bigcup_{I \in i} subSet(I)$.*

Proof. This could be directly concluded from Theorem 2. □

Theorem 4. *For a t-way interaction I, the additional test case is $T = addTF(I)$. In subinteraction set of T, the number of interactions that do not belong to $subSet(I)$ is $2^n - 2^t$.*

Proof. From Definition 4 we know that $|\text{subSet}(T)| = 2^n - 1$ and $|\text{subSet}(I)| = 2^t - 1$, in which I is the subinteraction of T. From Theorem 1 we know that $\text{subSet}(I) \subset \text{subSet}(T)$. So $|\text{subSet}(I) - \text{subSet}(T)| = 2^n - 2^t$. □

3. comFIL Algorithm

3.1. Description of comFIL. Theorem 3 and Definition 5 provide a screening method for obtaining the set of all minimal fault interactions from the set of candidate fault interactions; we name the set as complete Fault Interaction Location and denote it as comFIL for short. When an interaction I proves to be the fault interaction, we delete all the parent interactions of I from canFIS except I, while we delete all the child interactions of I from canFIS except I when it proves not to be the fault interaction.

Under the three assumptions in Section 2, algorithm comFIL is based on Theorems 2 and 3 and Definition 6 as theoretical core. According to Theorem 2, we calculate the set of candidate fault interactions from the combination test cases. And, based on Theorem 3 and Definition 6, we screen out the set of minimal fault interactions from the set of candidate fault interactions.

The basic framework of comFIL algorithm is shown in Algorithm 1. It has two input parameters CA and $R(T)$ and an output parameter canFIS, which is the minimal fault interaction set. Generally the algorithm process could be divided into 2 phases.

Generate the canFIS for fault location. Firstly, the algorithm counts the number of times that the fault interaction exists in passed or failed test cases separately (steps (2) to (12)); then the canFIS is screened out from test case set CA (steps (13) to (15)).

Generate the minimal fault interaction set (steps (16) to (20)). Steps (17) and (18) describe the following: if a schema I is a fault interaction, then we delete all its superinteraction except for I in canFIS. Steps (19) and (20) describe the following: if an interaction is not a fault interaction, then we delete all elements in its subinteraction in canFIS.

3.2. Implementation of Key Functions of Algorithm comFIL

(1) Implementation of subSet. The input of function subSet is an interaction I, and its output is the subinteraction set of I. For n-way interaction $I_n = \{v_1, v_2, v_3, \ldots, v_n\}$, its t-way subinteraction is a binary string $(b_1, b_2, b_3, \ldots, b_n)$, in which $\sum_{i=1}^{n} b_i = t$; that is, a t-way subinteraction of I_n is $\{v_{i_1}, v_{i_2}, v_{i_3}, \ldots, v_{i_t}\} = \{v_{i_j} \mid v_{i_j} \in I_n \wedge b_{i_j} = 1\}$, in which $i_1, i_2, i_3, \ldots, i_t$ are not equal to each other. For example, a subinteraction of interaction $\{v_1, v_2, v_3\}$ is $\{v_2, v_3\}$, whose binary string is (011).

A binary string is corresponding to a certain decimal number, so, for n-way interaction I_n, its subinteraction set is the corresponding binary stings of integer set $[1, 2^n - 1]$.

(2) Implementation of addTF. The input of function addTF is an interaction I, and its output is an additional test case T. For the input I, the number of its ways is not n; that is, I is not a test case. From Assumption 3 we can know that $I \subset T$; moreover, I is the only minimal fault interaction among all subinteractions of T.

From Theorem 4 we know that when generating addition test case T, we will first ensure that there is no minimal fault interaction in $2^n - 2^t$ subinteractions. However, if n is very large and t is relatively small, the numbers of interactions to be examined and additional test cases to be generated are very large. For convenience, we assume each parameter has a value, which does not associate with any fault (i.e., this value does not belong to any minimal fault interaction). This value is denoted as safe value.

In comFIL, the number of times that each interaction appears in passed test cases and failed test cases needs to be recorded. For a one-way interaction, that is, a value of a parameter, the value $f/(f + p)$ is named the fault ratio of the value. We simply consider that the smaller the fault ratio of a parameter's value is, the more likely this value would be the safe value of the parameter.

When it comes to testing an object as we could not get the safe value of each parameter before testing, according to Theorem 4, normally there will be an enormous number of interactions to be tested for the additional test cases generated by the interaction, resulting in the fact that the cost is very high. However, when there only exists a small amount of software defects (e.g., the software in the Delta testing

```
Inputs: CA: test case set
        R(T): test result
Output: canFIS: the minimal fault interaction set of CA.
Process:
(1)  Set canFIS = Φ, AllSet = Φ
                //canFIS save Candidate Fault Interaction Set;
                //AllSet save elements of CA, each element has 2 properties p, f.
        //phase I, Generate Candidate Fault Interaction Set
(2)     for (each_test_cases_T_in_CA){
(3)     if (R(T) == fail){
(4)             for (each_element_i_in_subSet(T)){
                                        //subSet() is a key function, we will explain it later.
(5)                     if (i ∉ AllSet){
(6)                             AllSet = AllSet ∪ {i}
                        }
(7)                     AllSet[i].f + +
                        //f is the number of test cases which include interaction and triggered system fault;
                }
(8)     }else{
(9)             for (each_element_i_in_subSet(T)){
(10)                    if (i ∉ AllSet){
(11)                            AllSet = AllSet ∪ {i}
                        }
(12)                    AllSet[i].p + +
                        //p is the number of test cases which include interaction but not trigger system fault;
                }
        }
(13)    for (each_element_i_in_AllSet){
(14)            if (i.p == 0){
(15)                    canFIS = canFIS ∪ {i}
                }
        }
        //phase II, Generate the minimal fault interaction set
(16)    while (there_are_element_i_not_tested_in_canFIS){
(17)            if (R(addTF(P1.IE)) == fail){
                        //addTF() is another key function, we will implement it later.
(18)                    canFIS = canFIS − {I | I ∉ canFIS, i ⊆ I}
(19)            }else{
(20)                    canFIS = canFIS − subSet(I)
                }
        }
(21)  return canFIS
```

Algorithm 1: The comFIL algorithm.

phase), the number of the minimal fault interactions in the combination test cases is very small and thus there is high possibility for the existence of safe value in the parameters.

The process of generating additional test cases is as follows.

First, if a value of a parameter p_i $(1 \le i \le n)$ appears in interaction I, the value of p_i in additional test case T is p_i itself; otherwise it will be assigned by the value that has the smallest safe value. If many values of p_i have the same smallest safe value, then p_i will be assigned randomly among these values.

Second, to check T, if it does not belong to test suite CA, then T is used as an additional test case; otherwise, T will be regenerated. The regeneration process is to modify a parameter's value in T by assigning this parameter another value whose safe value is the smallest or second smallest, thus making sure I is a subinteraction of T. Then we repeat this process till T does not belong to CA.

3.3. An Application of comFIL. As shown in Algorithm 2, foo is a method proposed in [13]. The correct prototype of foo in line (11) should be "$r+ = (b - d)/(a + 2)$"; because of the loss of "+" it becomes the wrong statement "$r = (b - d)/(a + 2)$," and it is a bug of program foo. The model of foo's input parameters is SUT (4; (2, 2, 3, 4)). The ranges of parameters a, b, c, d are $D(a) = \{0, 1\}$; $D(b) = \{0, 1\}$; $D(c) = \{0, 1, 2\}$; $D(d) = \{0, 1, 2, 3\}$, respectively. If an input contains $\{a.0, c.0\}$ or $\{a.0, c.3\}$, data stream could reach the

```
(1)  public static int foo (int a, int b, int c, int d)
(2)  {
(3)      int r = 1;
(4)      b+ = a + c;
(5)      switch (a)
(6)      {
(7)          case 0:
(8)              if (c < 1‖d > 2)
(9)
(10)                 //should be: r+ = (b − d)/(a + 2);
(11)                 r = (b − d)/(a + 2);
(12)             else
(13)                 r = b/(c + 2);
(14)             break;
(15)         case 1:
(16)             r = c ∗ (a − d);
(17)             break;
(18)     }
(19)     return r;
(20) }
```

Algorithm 2: A fault program.

Table 2: Test result of 2-way coverage.

Test #	a	b	c	d	Result
1	0	0	0	0	Fail
2	1	1	1	0	Pass
3	0	1	2	0	Pass
4	1	0	0	1	Pass
5	0	0	1	1	Pass
6	1	1	2	1	Pass
7	0	1	0	2	Fail
8	1	0	1	2	Pass
9	0	0	2	2	Pass
10	0	1	0	3	Fail
11	1	0	1	3	Pass
12	1	0	2	3	Pass

bug of foo; meanwhile, the state of foo is different from its correct prototype. Therefore, the minimal fault interaction set is $\{\{a.0, c.0\}, \{a.0, c.3\}\}$.

The program foo is tested on 2-way coverage in this paper. Test case set and test result are shown in Table 2. The algorithm comFIL uses Table 2 as input. The first process is to screen out canFIS from Table 2.

Table 3 shows the results of first process, in which the first column refers to the number of interactions while the first row represents the parameter of each interaction and the remaining rows represent the respective value of the parameters. Each row in Table 3 indicates an interaction from row 2 on. For example, interaction 1 $\{a.0, b.0, c.0\}$ is shown in row 2 (I # 1).

The second process is to generate minimal fault interaction set, as shown in Table 4. The second column of Table 4 describes the interactions contained by canFIS in each step.

Table 3: The canFIS of CA.

I #	a	b	c	d
1	**0**	0	**0**	—
2	**0**	0	—	0
3	**0**	—	**0**	0
4	—	0	**0**	0
5	**0**	—	**0**	—
6	—	0	—	0
7	—	—	**0**	0
8	**0**	0	**0**	0
9	**0**	1	**0**	2
10	**0**	1	**0**	—
11	—	1	**0**	—
12	**0**	—	**0**	2
13	—	—	**0**	2
14	—	1	—	2
15	—	1	—	2
16	**0**	1	—	2
17	**0**	1	**0**	3
18	**0**	—	**0**	3
19	—	—	**0**	3
20	**0**	—	—	3
21	—	1	—	3
22	—	1	**0**	3
23	**0**	1	—	3

The third column in Table 4 shows the interactions under testing. The fourth column shows the additional test cases for undertesting interactions. Columns 5 and 6 show the outputs of additional test cases and the set consisting of the elements deleted from canFIS, respectively.

Table 4: The step of computing canFIS.

Step #	canFIS	i	T = addTF(i)	$R(T)$	Delete from canFIS
(1)	$\{1, 2, \ldots, 23\}$	1	(0, 0, 0, 1)	Fail	{8}
(2)	$\{1, 2, \ldots, 7, 9, 10, \ldots, 23\}$	2	(0, 0, 1, 0)	Pass	{2, 6}
(3)	$\{1, 3, 4, 5, 7, 9, 10, \ldots, 23\}$	3	(0, 1, 0, 0)	Fail	∅
(4)	$\{1, 3, 4, 5, 7, 9, 10, \ldots, 23\}$	4	(1, 0, 0, 0)	Pass	{4, 7}
(5)	$\{1, 3, 5, 9, 10, \ldots, 23\}$	5	(0, 1, 0, 1)	Fail	{1, 3, 9, 10, 12, 17, 18}
(6)	{5, 11, 13, 14, 15, 16, 19, 20, 21, 22, 23}	11	(1, 1, 0, 1)	Pass	{11}
(7)	{5, 13, 14, 15, 16, 19, 20, 21, 22, 23}	13	(1, 1, 0, 2)	Pass	{13}
(8)	{5, 14, 15, 16, 19, 20, 21, 22, 23}	14	(1, 1, 1, 2)	Pass	{14}
(9)	{5, 15, 16, 19, 20, 21, 22, 23}	15	(1, 1, 0, 2)	Pass	{15}
(10)	{5, 16, 19, 20, 21, 22, 23}	16	(0, 1, 1, 2)	Pass	{16}
(11)	{5, 19, 20, 21, 22, 23}	19	(1, 1, 2, 3)	Pass	{19}
(12)	{5, 20, 21, 22, 23}	20	(0, 1, 1, 3)	Fail	{23}
(13)	{5, 20, 21, 22}	21	(1, 1, 2, 3)	Pass	{21}
(14)	{5, 20, 22}	22	(1, 1, 0, 3)	Pass	{22}
(15)	{5, 20}	—	—	—	—
(16)	$\{\{a.0, c.0\}, \{a.0, d.3\}\}$	—	—	—	—

We can conclude from Table 4 that the whole process takes 14 steps and each step generates an additional test case. The minimal fault interaction set $\{5, 20\}$ is screened out at step (15) at last. Meanwhile we can get a conclusion that the number of steps the process takes depends on the order of interactions being tested. For example, if, in step (13) test interaction 22, the element to delete in canFIS is $\{21, 22\}$, then the minimal fault interaction set could be generated directly. The whole process only takes 13 steps and needs only 13 test cases. Therefore, an optimized interaction test order could reduce the generating of test cases. In this paper, we do not carry on the discussion and simply consider the order is random. The method presented by Ghandehari et al. [13] give the result that contains 9 interactions in the set; however, the minimal fault interaction set contains 2 interactions. This shows that the method comFIL is more precise than the method proposed by Ghandehari et al.

4. Empirical Study

In algorithm comFIL, we need to record the number p of each interaction existing in passed test cases and the number f of each interaction existing in failed test cases. For a 1-way interaction, the value of $f/(f + p)$ is called fault ratio. The smaller the value is, the more probable the value is a safe value of the parameter.

4.1. Additional Test Case Generation. In our experiment, the additional test case generation follows the following two steps:

(I) For an input parameter p_i, $i \in [1, n]$, if its value is in interaction I, then the value of p_i in T has the same value; otherwise, p_i in T is assigned by its value that has the smallest fault ratio.

(II) Check whether T belongs to CA; if not, then T is used as additional test case; otherwise, change a value of p_i in T till T does not belong to CA.

4.2. Evaluation Criteria. We use fault ratio as our evaluation criteria. In comFIL algorithm, we need to record the number p of each interaction that exists in passed test cases and the number f of each interaction that exists in failed test cases. For a 1-way interaction, the value of $f/(f + p)$ is called fault ratio. The smaller the value is, the more probable the value will be a safe value of the parameter.

4.3. Test Oracle. Since the feature of these programs is not a concern in this paper, they are assumed to be correct. Then the standard and fault versions are compiled and run with test case T as input; if the outputs of standard and fault versions are different, we believe the test case triggers the fault; that is, $R(T)$ = fail; otherwise, $R(T)$ = pass.

4.4. Experiment 1

(1) Experiment Objective. We use six C programs (comdline, count, nametbl, ntree, series, and tokens [12]) as test samples and input parameter model presented by Z. Zhang and J. Zhang [12]. Table 5 shows the basic information of these programs.

The second column represents the number of lines without comments in each program, while column 3 refers to their input models. For example, comdline (9; $(2^1, 3^4, 4^1, 6^2, 15^1)$) means comdline has 9 parameters, in which 4 parameters have 3 values, 2 parameters have only 1 value, 1 parameter has 2 values, 1 parameter has 4 values, 2 parameters have 6 values, and 1 parameter has 15 values. Count (6; (2, 2, 3, 3, 3, 3)) can also be represented as count (6; $(2^2, 3^4)$).

Table 5: Test sample.

Program	Number of lines	Input model
Comdline	42	Comdline (9; (2^1, 3^4, 4^1, 6^2, 15^1))
Count	288	Count (6; (2^2, 3^4))
Nametbl	129	Nametbl (8; (2^4, 3^2, 5^2))
Ntree	307	Ntree (6; (2^2, 4^4))
Series	329	Series (4; (2^1, 4^2, 6^1))
Tokens	336	Tokens (4; (2^2, 3^2))

(2) Result of the Experiment. Table 6 indicates the detailed test results of the experiment; the data is mainly focused on test steps (additional test cases) and radix. In Table 6, column 2 shows the number of test cases. Column 3 represents the different fault versions of each program. Column 4 shows the size of each canFIS. Column 5 refers to the number of test cases needed by each canFIS. Columns 6~11 represent the number of x-way minimal fault interactions and the number of x-way fault interactions to be selected, respectively; x could be identified by the column title. For example, the first fault version of comdline is shown in Table 6. It means that its canFIS's size is 1663, and it needs 149 test cases. The numbers of 1~5-way minimal fault interactions are 1; 0; 0; 0; 0, respectively, and the minimal fault interaction larger than 5 ways is 0. The numbers of interactions being tested for computing each canFIS are 1; 15; 20; 22; 27; 64, respectively.

Table 7 is an experiment result compared with FIC algorithm presented by Z. Zhang and J. Zhang [12].

Table 7 shows in most situations comFIL could generate a minor minimal fault interaction set compared to FIC. However, to locate the minimal fault interaction in software, comFIL need the involvement of more additional test cases than FIC algorithm. The main cause for that is that, for a test case that triggers the fault, it has $2^n - 1$ child interaction; if the number of minimal fault interactions covers the array and test cases, we can not exclude the idea that most interactions of the child interaction set is fault interaction during the initial interaction fault detection.

We may find that the result of ntree is not the minimal fault interaction set of CA. That is because many "assert" statements exist in its source codes. These statements make the program exit before data stream reaches the bug; that is, the program does not satisfy Assumption 2. Therefore, the comFIL algorithm failed to identify the minimal fault interaction set of ntree.

4.5. Experiment II. Generally, we consider a technique more effective if it could generate fewer test cases and be highly active. In this section, in order to verify the complexity of comFIL and show the additional test case count needs in different testing models, two simulated experiments will be examined. All the programs are developed with special design in order to better match the 3 assumptions of comFIL.

(1) Simulated Experiment 1. In this simulated experiment, we developed a simple program, Animal; it has an input model SUT(7; (5^7)); that is, the system has 7 parameters, and each of them has 5 values. We produce 5 fault versions by injecting fault in different position; each has 2~6 minimal fault interactions. The result of the experiment is shown in Figure 1. The histogram in red shows the radix of canFIS while the data in green indicates the steps (additional test cases) to take to screen out the canFIS.

(2) Simulated Experiment 2. We developed 6 different programs which are much more complex than Animal in experiment 1; their input models are SUT(5; (5^6)); SUT(6; (5^6)); SUT(7; (5^7)); SUT(8; (5^8)); SUT(9; (5^9)); SUT(10; (5^{11})), respectively. Each program has 5 minimal fault interactions. The result of the experiment is shown in Figure 2. Figure 2(a) shows the ratio of the number of additional test cases to the radix of canFIS. Figure 2(b) shows the result of experiment 2.

The simulated experiment result presented by Figures 1 and 2 shows the number of additional test cases is decreasing while the program could better match the 3 assumptions of this paper. Furthermore, with the increment of minimal fault interactions' count, the Ratio becomes smaller; this means the higher probability of getting safe value for input parameters. However, when the minimal fault interaction increases to a special value such as 9 in Figure 2(b), the Ratio would rise; this is because when the number of input parameters becomes too large, too many additional test cases will be generated; this would affect the efficiency and lead the Ratio to rise.

4.6. Experiment Conclusion. We can get the following conclusion about comFIL algorithm from both real program and simulated experiment results.

(1) comFIL Could Obtain a Minimal Fault Interaction Set. From the result of experiment I—Table 7, we can see that comFIL could generate a minor minimal fault interaction set compared to FIC.

(2) comFIL Has Higher Capability of Getting Safe Value for Parameters. The result of experiment II shows, with the increment of minimal fault interactions' count, the Ratio becomes smaller; this means the higher probability of getting safe value for input parameters.

(3) The Efficiency of comFIL Would Be Affected While the Number of Input Parameters Is Too Large. When the number of input parameters is too large, the additional test cases size would be very huge; this is a limitation of comFIL, but we found that [14] presents an approach of leveraging the notion of inducing combination to locate the faults inside the source code; the combination of these two approaches would potentially benefit combinatorial testing.

(4) The Assumptions Are Common but Sometimes They Do Not Establish in Real-World Programs. Although the assumptions of comFIL are common in combination testing study area, but sometimes they do not establish in real-world programs; this would limit the application of the algorithm and need to be further studied.

Table 6: Test result of standard program.

Program	CA size	ver	canFIS radix	canFIS steps	1	2	3	4	5	Over 5
Comdline	95	1	1663	149	1	0	0	0	0	0
					1	15	20	22	27	64
		2	10734	2033	0	0	50	28	0	0
					0	6	219	415	613	780
		3	2392	284	25	29	**87**	24	82	0
					1	25	29	87	24	82
Count	12	1	121	36	0	2	0	0	0	0
					0	2	19	14	1	0
		2	250	89	0	**4**	0	0	0	0
					0	5	41	24	19	0
Nametbl	25	1	23	12	0	0	2	0	0	—
					0	1	6	5	0	—
		2	109	54	0	**7**	4	0	0	—
					0	9	31	14	0	—
		3	41	20	0	0	3	0	0	—
					0	0	9	11	0	—
Ntree	16	1	47	43	0	**11**	1	0	—	—
					0	32	11	0	—	—
		2	66	48	0	3	0	0	—	—
					0	32	16	0	—	—
Series	24	1	18	15	0	0	2	0	—	—
					0	7	8	0		
		2	57	34	0	**13**	0	0	—	—
					0	19	17	0	—	—
Tokens	3	1	23	13	**1**	0	0	0	—	—
					1	9	3	0	—	—
		2	23	10	1	0	0	0	—	—
					1	6	4	0	—	—

Table 7: Compare with FIC.

Program	Radix of comFIL (fault interaction)	# fault interaction of FIC	Test steps of comFIL	# Adaptive tests of FIC
Comdline	87	10	219	0
Count	4	10	41	14
Nametbl	7	22	31	62
Ntree	11	13	32	16
Series	13	5	19	26
Tokens	1	5	9	12

5. Conclusion and Future Work

5.1. Conclusion. In this paper, we present a new combinatorial testing algorithm named comFIL, which could screen out the minimal fault interaction set of test cases.

The main contributions of this paper are listed as follows:

(1) Summarizing the basic idea of the previous fault interaction location techniques, including their advantages and disadvantages.

(2) Proposing a novel fault interaction location method named comFIL (complete Fault Interaction Location) which has more powerful functionalities and performs more precisely compared with other fault location techniques.

(3) Presenting the basic idea and theory model for comFIL and illustrating 2 key points when implementing comFIL.

(4) Using 6 programs as sample and 2 additional simulated experiments to verify the precision and effectiveness of comFIL.

If we can not obtain the safe value of each parameter before testing, the cost in generating an additional test case for an interaction is very high. However, if there are only a few bugs in a program, the number of minimal fault interactions

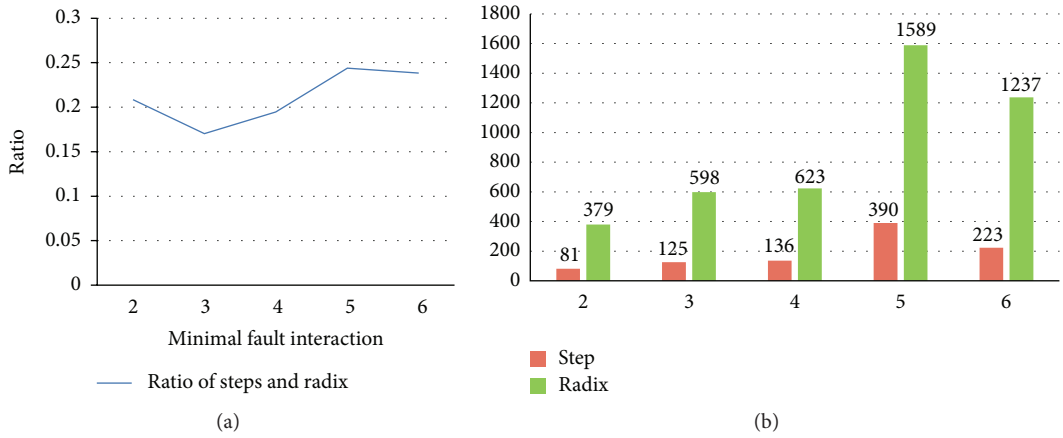

Figure 1: Test result of Simulated Experiment 1. (a) Ratio of steps and radix. (b) Step and radix.

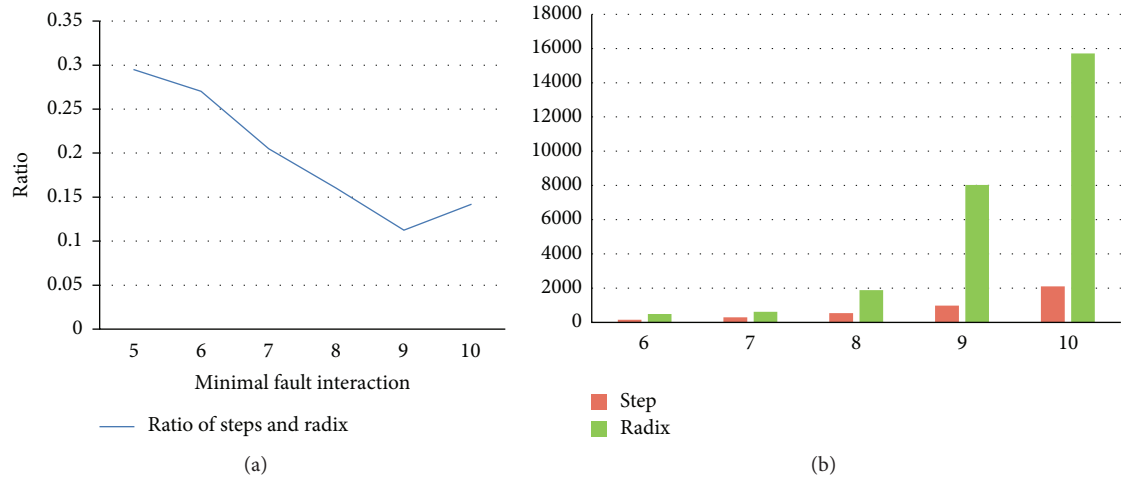

Figure 2: Test result of Simulated Experiment 2. (a) Ratio of steps and radix. (b) Step and radix.

is small and it is more probable for a parameter to be in its safe value. When we calculate the safe values of each parameter with proper methods, it is almost impossible that generated additional test cases do not satisfy Assumption 3. Even for a test case generated for an interaction randomly, its possibility that it does not satisfy Assumption 3 is rather low. So almost every testing method in combinatorial testing could only work effectively when applied in program with relatively less faults.

The theory and experiments indicate that comFIL is more accurate in fault localization compared with other algorithms in terms of combinatorial testing. However, comFIL also has its deficiency: (1) Assumption 3 is very strong and (2) the number of additional test cases to be generated is very large.

5.2. Future Work. The future work will include three aspects: (1) for algorithm comFIL, lots of additional test cases should be generated; however, the order of the interactions being tested will influence the number of additional test cases generated. Thus optimizing the order of interactions being tested to reduce the number of additional test cases will be an interesting direction to explore further in the future. (2) The objective of algorithm comFIL is to generate the minimal fault interaction set, while how to use the minimal fault interaction set to further locate bugs will also be a significant topic to study further afterwards. (3) Exploring more effective combinatorial testing method according to different type of software (such as web service) is also worthwhile of further study.

Competing Interests

The authors declare that they have no competing interests.

References

[1] J. D. Hagar, T. L. Wissink, D. R. Kuhn, and R. N. Kacker, "Introducing combinatorial testing in a large organization," *Computer*, vol. 48, no. 4, pp. 64–72, 2015.

[2] W.-J. Zhou, D.-P. Zhang, and B.-W. Xu, "Locating error interactions based on partial covering array," *Chinese Journal of Computers*, vol. 34, no. 6, pp. 1126–1136, 2011.

[3] C. Nie and H. Leung, "The minimal failure-causing schema of combinatorial testing," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 4, article 15, 2011.

[4] B. Garn and D. E. Simos, "Eris: a tool for combinatorial testing of the Linux system call interface," in *Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW '14)*, pp. 58–67, Cleveland, Ohio, USA, April 2014.

[5] S. K. Khalsa and Y. Labiche, "An orchestrated survey of available algorithms and tools for combinatorial testing," in *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering (ISSRE '14)*, pp. 323–334, IEEE, Naples, Italy, November 2014.

[6] A. Gupta and C. H. Scholz, "A model of normal fault interaction based on observations and theory," *Journal of Structural Geology*, vol. 22, no. 7, pp. 865–879, 2000.

[7] D. C. P. Peacock, "Propagation, interaction and linkage in normal fault systems," *Earth-Science Reviews*, vol. 58, no. 1-2, pp. 121–142, 2002.

[8] C. J. Colbourn and D. W. McClary, "Locating and detecting arrays for interaction faults," *Journal of Combinatorial Optimization*, vol. 15, no. 1, pp. 17–48, 2008.

[9] C. Martínez, L. Moura, D. Panario et al., "Algorithms to locate errors using covering arrays," in *LATIN 2008: Theoretical Informatics*, pp. 504–519, Springer, Berlin, Germany, 2008.

[10] J. Piton-Gonçalves and S. M. Aluísio, "An architecture for multidimensional computer adaptive test with educational purposes," in *Proceedings of the 18th Brazilian symposium on Multimedia and the web (WebMedia '12)*, pp. 17–24, ACM, São Paulo, Brazil, 2012.

[11] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[12] Z. Zhang and J. Zhang, "Characterizing failure-causing parameter interactions by adaptive testing," in *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA '11)*, pp. 331–341, ACM, Toronto, Canada, July 2011.

[13] L. Ghandehari, J. Chandrasekaran, Y. Lei, R. Kacker, and D. Kuhn, "Short paper: BEN: a combinatorial testing-based fault localization tool," in *Proceedings of the 4th International Workshop on Combinatorial Testing (in Junction with 8th IEEE International Conference on Software Testing, Verification and Validation)*, April 2015.

[14] L. S. Ghandehari, Y. Lei, D. Kung, R. N. Kacker, and D. R. Kuhn, "Fault localization based on failure-inducing combinations," in *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering (ISSRE '13)*, pp. 168–177, Pasadena, Calif, USA, November 2013.

[15] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 129–139, ACM, 2007.