Characterizing Failure-Causing Parameter Interactions by Adaptive Testing*

Zhiqiang Zhang
State Key Laboratory of Computer Science,
Institute of Software,
Chinese Academy of Sciences
Graduate University,
Chinese Academy of Sciences
zhangzq@ios.ac.cn

Jian Zhang
State Key Laboratory of Computer Science,
Institute of Software,
Chinese Academy of Sciences
zj@ios.ac.cn

ABSTRACT

Combinatorial testing is a widely used black-box testing technique, which is used to detect failures caused by parameter interactions (we call them *faulty interactions*). Traditional combinatorial testing techniques provide fault detection, but most of them provide weak fault diagnosis. In this paper, we propose a new fault characterization method called *faulty interaction characterization* (`FIC`) and its binary search alternative `FIC_BS` to locate one failure-causing interaction in a single failing test case. In addition, we provide a tradeoff strategy of locating multiple faulty interactions in one test case. Our methods are based on adaptive black-box testing, in which test cases are generated based on outcomes of previous tests. For locating a t-way faulty interaction, the number of test cases used is at most k (for `FIC`) or $t(\lceil \log_2 k \rceil + 1) + 1$ (for `FIC_BS`), where k is the number of parameters. Simulation experiments show that our method needs smaller number of adaptive test cases than most existing methods for locating randomly-generated faulty interactions. Yet it has stronger or equivalent ability of locating faulty interactions.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Testing tools*

General Terms

Algorithms, Reliability

Keywords

Combinatorial testing, diagnostics, faulty interaction, adaptive testing, group testing

*This work is supported in part by the National Science Foundation of China (Grant No. 61070039) and the High-Tech (863) program of China (Grant No. 2009AA01Z148).

ISSTA'11, July 17–21, 2011, Toronto, ON, Canada

1. INTRODUCTION

Assume that we are testing the functionality of a software whose behavior is affected by k parameters (or called "factors"), an ideal plan is to apply exhaustive testing in which we test all possible combinations of the parameters. However, the number of test cases of exhaustive testing grows exponentially with k. Suppose each parameter has two possible values, then we need 2^k test cases. So it is impossible to be applied on many large applications.

Combinatorial testing is a widely used black-box testing technique, which uses *covering arrays* (CA) or *mixed covering arrays* (MCA) [2] as the test suite to detect failures caused by parameter interactions. A $CA(n; k, t, s)$ is an array of n rows and k columns, where n is the size of the CA, k is the number of parameters and s is the number of possible values of each parameter. Each column represents a parameter and each row represents a test case. The CA is of strength t, i.e. for any t columns of the array, the sub-array covers all possible combinations of the corresponding t parameters. MCA is similar to CA, while the only difference is that parameters may have different number of possible values.

The fault model of combinatorial testing assumes that failures are caused by parameter interactions. A t-way interaction or an interaction of size t is an assignment of some specific value to each parameter of selected t parameters. A failure-causing interaction is called a *faulty interaction*. Kuhn and Reilly's investigation on software failures [6] stated that failures are always caused by interactions of small sizes. Although it is not universally true, we mainly focus on this kind of failures in this paper. Since a covering array of strength t covers all t-way interactions, we can detect all failures caused by faulty interactions of size no more than t. Moreover, the size of CAs grows logarithmically with k [1]. Thus, combinatorial testing is quite resource-saving in practical applications.

After a failure is detected by applying combinatorial testing, we know that it is caused by a faulty interaction. The failure is actually caused by a bug inside the software. And we shall locate the bug before fixing it. There are massive research on combinatorial testing in recent years [11]. However, most traditional combinatorial testing techniques aim at increasing interaction coverage and reducing the number of test cases, while very few of them focus on diagnosis.

Table 1 shows a covering array of strength 2 for testing an online payment system. The interactions in braces ("{}")

Table 1: A Sample Covering Array

Client	Web Server	Payment	Database	Exec
Firefox	WebSphere	MasterCard	DB/2	`pass`
Firefox	.NET	UnionPay	Oracle	`pass`
Firefox	{Apache}	{Visa}	Access	`fail`
IE	WebSphere	UnionPay	Access	`pass`
{IE}	Apache	MasterCard	{Oracle}	`fail`
IE	.NET	Visa	DB/2	`pass`
Opera	WebSphere	Visa	Oracle	`pass`
Opera	.NET	MasterCard	Access	`pass`
Opera	Apache	UnionPay	DB/2	`pass`

are faulty interactions, and the test results are shown in the last column. We can see that this covering array can detect all faulty interactions of strength 2, but we cannot tell what the faulty interactions are from the test results. Now our goal is to discover the faulty interactions in failing tests. Such diagnostic information can be used to minimize the configuration space of failures, and thus can help us locate bugs in the software.

Contributions of this paper: We introduce a new fault characterization method called *faulty interaction characterization* (`FIC`) and its binary search alternative `FIC_BS` to *locate* (means to discover or find out) one faulty interaction in a single failing test case. In addition, we provide a tradeoff strategy of locating multiple faulty interactions in one test case. Our methods are based on adaptive testing, in which we use a failing test case as the *seed test case*, and adaptively generate and execute some additional test cases based on it. Eventually, our methods can locate some faulty interactions in the seed test case. Those additional test cases are called *adaptive test cases*. For locating a t-way faulty interaction in the seed test case, the number of adaptive test cases is at most k (for `FIC`) or $t(\lceil \log_2 k \rceil + 1) + 1$ (for `FIC_BS`), where k is the number of parameters. Simulation experiments show that our method needs smaller number of adaptive test cases than most existing methods for locating randomly-generated faulty interactions. Yet it has stronger or equivalent ability of locating faulty interactions.

The rest of this paper is organized as follows: We introduce the definitions and notations used in this paper in Section 2, and describe our methods and illustrate how they work in Section 3. In Section 4, we talk about the assumptions we made in this paper. Related works are discussed in Section 5. Experimental results are shown in Section 6. In Section 7, we give the conclusions and discuss future work. Appendix A is a proof for the correctness of `FIC`.

2. PRELIMINARIES

We assume that the software under test (SUT) has k input parameters. Then we generate and execute test cases on the SUT. A test case is an assignment to all of the k parameters. Combinatorial testing aims at generating tests to reveal *interaction faults*.

Now we give the definitions used in this paper:

Definition 1. An $SUT(k; s_1, s_2, \ldots, s_k)$ has k parameters ($k > 0$). The value of the i^{th} parameter $v_i \in \{1, 2, \ldots, s_i\}$, where $s_i > 0$ for $i \in \{1, 2, \ldots, k\}$. k is the *number of parameters*, and s_i is the *level of parameter* v_i.

We may represent an $SUT(k; s_1, s_2, \ldots, s_k)$ in its short form $SUT(k; s_{i_1}^{a_1} s_{i_2}^{a_2} \ldots)$. All s_{i_j}'s differ from each other, indicating the levels of parameters, while $1 \le s_{i_1} < s_{i_2} < \cdots$. The exponents a_j indicate the number of parameters at level s_{i_j}, where $j = 1, 2, \ldots$. For example, $SUT(4; 2, 2, 2, 3)$ can be represented as $SUT(4; 2^3 3^1)$.

Definition 2. Given an $SUT(k; s_1, s_2, \ldots, s_k)$, a *test case* is a k-tuple $V = (v_1, v_2, \ldots, v_k)$, where $v_i \in \{1, 2, \ldots, s_i\}$, for $i \in \{1, 2, \ldots, k\}$.

In this paper, $\hat{v}_i$ denotes the ith parameter, and $val(V, \hat{v}_i)$ denotes its value in test case V.

Many works in faulty interaction characterization use various denotations for interactions, here we use the following notation which is the same with Nie's paper [12] (they use the term "schemas" for interactions).

Definition 3. Given $SUT(k; s_1, s_2, \ldots, s_k)$, an *interaction of size t* or a *t-way interaction* is a k-tuple $p = (p_1, p_2, \ldots, p_k)$ having t *fixed parameters* and $k - t$ *free parameters*, where $t \ge 0$ and

$$p_i = \begin{cases} s_{j_i} \in \{1, 2, \ldots, s_i\}, & \text{if } \hat{v}_i \text{ is a fixed parameter,} \\ \text{"-"}, & \text{if } \hat{v}_i \text{ is a free parameter.} \end{cases}$$

If all fixed parameters of an interaction p have exactly the same values in a test case V, then we say V *matches* p, or p is an interaction of V. It is easy to see that a test case contains 2^k interactions. An interaction that can cause failures is called a *faulty interaction*.

Suppose p is an interaction, denote $P_{fixed}(p)$ as the set of all fixed parameters of p, and $P_{free}(p)$ as the set of all free parameters of p, thus $P_{free}(p) = \{\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_k\} \setminus P_{fixed}(p)$.

Suppose V is a test case, then we denote $\Gamma(V)$ as the set of all faulty interactions of V. When we are discussing only the interactions of a given test case V, the interactions' *vector form* in Definition 3 and their *set form* represented by P_{free}'s are freely exchangeable.

Suppose $p1$ and $p2$ are two interactions of V, We say that $p1$ and $p2$ are *overlapping* if $P_{fixed}(p1) \cap P_{fixed}(p2) \neq \emptyset$. If $P_{fixed}(p1) \subseteq P_{fixed}(p2)$, then we say $p1$ is a sub-interaction of $p2$ ($p1 \sqsubseteq p2$) and $p2$ is a super-interaction of $p1$ ($p2 \sqsupseteq p1$).

A basic operation of `FIC` is to *modify* parameter values. Here "modify" means to change a parameter's original value to an arbitrarily different one within its domain. Modifying parameter values can activate or deactivate some interactions.

Definition 4. Suppose V is a test case, $S = (s_1, s_2, \ldots, s_k)$ specifies the levels of parameters and U is a set of parameters, then we use $\mathcal{M}(V, S, U)$ to denote a new test case after modifying all parameters in U of V.

Example 1. Suppose we have an $SUT(4; 2^4)$ and a failing test case V=`(1,2,2,1)`, and a failure happens when $v_1 = 1$ and $v_3 = 2$, then we say V has a faulty interaction p =`(1, -,2,-)` or $\{\hat{v}_1, \hat{v}_3\}$. It is deactivated after we modify $\hat{v}_1$ of `(1,2,2,1)` from 1 to 2, while it is activated after we modify $\hat{v}_1$ of `(2,2,2,1)` from 2 to 1.

Let $U = \{\hat{v}_1, \hat{v}_3\}$ and $S =$ `(2,2,2,2)`, then we have $\mathcal{M}(V, S, U)$=`(2,2,1,1)`.

3. LOCATING FAULTY INTERACTIONS

In this section, we will introduce our new fault characterization methods: `FIC` and `FIC_BS`. We use each failing test

case as a seed test case V_s, and adaptively generate and execute some new test cases by modifying parameter values of V_s. Eventually, we can locate the faulty interactions of the seed test case.

3.1 Assumptions

Locating faulty interactions is a hard problem. A seed test case of size k may have 2^k possible faulty interactions. Many previous works in faulty interaction characterization have made several assumptions to make clear studies, here we write them down explicitly. These assumptions help us to discuss upon an ideal problem framework. However, these assumptions may seem to be somehow strong and do not hold in many practical applications. In the later part of this paper, we will discuss how to weaken these assumptions. That is, to resolve the inconsistency between real software and the ideal problem framework.

ASSUMPTION 1. *The outcome of executing a test case on the SUT is either* `pass` *or* `fail`.

In some testing applications the SUT fails with some failure types, indicating what types of failure occurred. Or sometimes, a test execution may produce an `unresolved` result, indicating that we cannot determine whether some specific failure occurs or not. For a clear discussion, we use `pass` and `fail` as test results in this paper.

ASSUMPTION 2. *Test cases matching a faulty interaction must fail.*

Sometimes a test case matching a faulty interaction may pass during testing due to coincidental correctness, or because of some errors' domination over others (e.g. Assume that there are multiple error types and we are characterizing faulty interactions for bad output error, the error may not present when the program crashes before making an output).

ASSUMPTION 3. *All parameters are independent.*

In some applications, there are constraints on parameter values, such as "if v_1=1 then v_3=2" or other kinds of constraints. Most similar techniques do not address this problem. For a clear discussion, we do not consider parameter constraints in this section.

ASSUMPTION 4. *All faulty interactions that appear during* `FIC` *are faulty interactions of V_s.*

This assumption ensures that no new faulty interactions are activated when we modify parameters in V_s during `FIC`. The appearance of new faulty interactions will increase the complexity of the problem. We avoid discussing this condition in this section.

Now our problem is converted into a generalized group testing problem [14]. All discussions in the rest of the section are based on the four assumptions above, if not explicitly stated.

3.2 Basic Idea

Here we introduce some conclusions to help us design a new algorithm for faulty interaction characterization. According to Assumption 4, all faulty interactions are of V_s.

Suppose we have a set of parameters U. We modify all parameters of V_s in U, then all faulty interactions containing any parameters in U will be deactivated, and the rest faulty interactions are still active. So we have the following lemma:

Algorithm 1 Faulty Interaction Characterization

```
1: function FIC(V_s,S,k,C_tabu)
2:     interaction ← ∅
3:     C_free ← C_tabu
4:     while true do
5:         (C_free, param) ← LocateFixedParam(
                V_s, k, S, C_free, interaction)
6:         if param = null then
7:             break
8:         interaction ← interaction ∪ {param}
9:     return interaction
```

Algorithm 2 Locating One Fixed Parameter

```
1: function LocateFixedParam(V_s,S,k,C_free,interaction)
2:     C_cand ← {v̂_1, v̂_2, ..., v̂_k} \ C_free \ interaction
3:     U ← C_free
4:     while C_cand ≠ ∅ do
5:         param ← Extract(C_cand)
6:         if Run(M(V_s, S, U ∪ {param})) = pass then
7:             return (U, param)
8:         U ← U ∪ {param}
9:     return (U, null)
```

LEMMA 1. *Suppose we have a faulty interaction $p \in \Gamma(V_s)$. Then $p \in \Gamma(\mathcal{M}(V_s, S, U))$ if and only if p contains only those parameters in $\{\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_k\} \setminus U$.*

Suppose we have two sets of parameters U_1 and U_2 and $U_1 \subseteq U_2$. Then all faulty interactions of $\mathcal{M}(V_s, S, U_2)$ contain only those parameters in $\{\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_k\} \setminus U_2$. And then those faulty interactions contain only those parameters in $\{\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_k\} \setminus U_1$, so they are also faulty interactions of $\mathcal{M}(V_s, S, U_1)$. We have the following lemma:

LEMMA 2. *Suppose we have two sets of parameters U_1, U_2 and $U_1 \subseteq U_2$, then $\Gamma(\mathcal{M}(V_s, S, U_1)) \supseteq \Gamma(\mathcal{M}(V_s, S, U_2))$.*

The following lemma is the essential lemma of `FIC`, it figures out how to locate a fixed parameter of a faulty interaction of V_s:

LEMMA 3. *Suppose we have a set of parameters U and a parameter $\hat{v}_i$, such that $\mathcal{M}(V_s, S, U)$ fails and $\mathcal{M}(V_s, S, U \cup \{\hat{v}_i\})$ passes, then $\hat{v}_i$ is a fixed parameter of all faulty interactions of $\mathcal{M}(V_s, S, U)$.*

PROOF. As $\mathcal{M}(V_s, S, U)$ fails, we know that it contains at least one faulty interaction. If $\hat{v}_i$ is not fixed parameter of one faulty interaction p of $\mathcal{M}(V_s, S, U)$, then p is still a faulty interaction of $\mathcal{M}(V_s, S, U \cup \{\hat{v}_i\})$, then the test case shall fail. □

The main idea of `FIC` is repetitively finding U and $\hat{v}_i$ satisfying the requirement in Lemma 3 in order to *locate* fixed parameters of a faulty interaction. Meanwhile, `FIC` maintains a list of free parameters (variable "C_{free}" in Algorithm 1), which ensures the correctness of the faulty interactions we located. Here "locate" a parameter or a faulty interaction means to discover or find out a fixed parameter or a faulty interaction. See the pseudo-code in Algorithm 1, 2. `FIC` is another form of the adaptive algorithm in [14].

In Algorithm 2, the auxiliary function *Extract*(A) in line 5 arbitrarily extracts and returns one element in A, and

function $Run(V)$ in line 6 executes test case V and returns the outcome (`pass` or `fail`).

Here we briefly explain the functions of `FIC`:

3.2.1 Function *FIC*

The function is used to locate a new faulty interaction containing only those parameters not in C_{tabu}. In this function, the input V_s is the seed test case, S is the vector of parameter levels, k is the number of parameters, and C_{tabu} is a tabu list of parameters that the new faulty interaction located by *FIC* shall not contain. The inputs must satisfy: $Run(\mathcal{M}(V_s, S, C_{tabu})) = \texttt{fail}$.

Suppose $interaction_{new}$ is the faulty interaction we are locating. The local variables are described as follows:

- *interaction* is the set of all located fixed parameters of $interaction_{new}$.
- C_{free} is the set of all the already-determined free parameters of $interaction_{new}$.
- *param* is a fixed parameter of $interaction_{new}$ newly-located by function *LocateFixedParam*.

As the new faulty interaction shall not contain parameters in C_{tabu}, C_{free} is initialized as C_{tabu}. In the while-loop, the function repetitively tries to locate new fixed parameters of $interaction_{new}$ (line 5) until no new fixed parameter is located (line 6-7). After the loop terminates, the function returns *interaction* as the newly-located faulty interaction.

3.2.2 Function *LocateFixedParam*

The function is used to locate the next fixed parameter of $interaction_{new}$. In this function, the input V_s, S, k are identical with *FIC*, C_{free} is the set of all the already-determined free parameters of $interaction_{new}$, *interaction* is the set of all located fixed parameters of $interaction_{new}$. The local variables are described as follows:

- C_{cand} is the set of candidate parameters which are possible to be the next fixed parameter of $interaction_{new}$.
- U is the set of the already-determined free parameters of $interaction_{new}$.
- *param* is a parameter extracted from C_{cand}.

This function is used to locate a new fixed parameter of V_s. It always modifies parameters in C_{free}, and always keeps parameters in *interaction* and ensures that all activated faulty interactions are super-interactions of *interaction*. In each cycle of the while-loop, the function modifies one more parameter compared with the previous cycle. If the test case passed, then U and *param* satisfy the conditions in Lemma 3. Variable U contains all already-determined free parameters of $interaction_{new}$.

3.2.3 Locating Multiple Faulty Interactions

Now we can see that the `FIC` algorithm can locate one faulty interaction in V_s. While sometimes V_s may contain multiple faulty interactions, and people may want to locate all of them. But in fact, this goal is not realistic, as the number of adaptive test cases may increase rapidly with the growth of the interconnection of faulty interactions. We want to locate more faulty interactions while considering the number of adaptive tests. Here we provide a tradeoff strategy of locating multiple faulty interactions in V_s. See Algorithm 3.

Algorithm 3 Locating Non-overlapping Faulty Interactions

```
1: function FINOVLP(V_s,S,k)
2:     interactions ← ∅
3:     C_tabu ← ∅
4:     while true do
5:         if Run(M(V_s, S, C_tabu)) = pass then
6:             break
7:         interaction ← FIC(V_s, S, k, C_tabu)
8:         interactions ← interactions ∪ {interaction}
9:         C_tabu ← C_tabu ∪ interaction
10:        if interaction = ∅ then
11:            break
12:    return interactions
```

Function *FINOVLP* locates non-overlapping faulty interactions of V_s. But this does not mean that the faulty interactions in V_s shall be non-overlapping. In this function, the inputs V_s, S and k are identical with function *FIC*. *interactions* is the set of all located faulty interactions. C_{tabu} contains all fixed parameters of located faulty interactions. In the while-loop, the function first runs a test case to determine whether there is another faulty interaction to locate (line 5-6). Then it tries to locate a new faulty interaction (line 7). The while-loop terminates when there is no new faulty interaction to locate (line 5-6) or the SUT always fail ($interaction = \emptyset$, line 10-11). Eventually, after the loop terminates, *FINOVLP* returns a set of non-overlapping faulty interactions (line 12).

3.2.4 An Example

Here is an example showing how `FIC` and `FINOVLP` work.

Example 2. Suppose we have an $SUT(8; 2^8)$, with a failing test case V_s = `(1,2,2,1,2,2,1,1)`. Two faulty interactions exist in V_s: p_1 = `(-,-,2,-,-,2,-,-)` = $\{\hat{v}_3, \hat{v}_6\}$ and p_2 = `(-,-,-,1,-,-,-,-)` = $\{\hat{v}_4\}$. Suppose we know nothing about the faulty interactions before running `FINOVLP` and `FIC`. Table 2 shows how the two algorithm works.

In Table 2, the parameters tagged "p_1" are fixed parameters of p_1, the parameters tagged "p_2" are fixed parameters of faulty interaction p_2. A *SKIP* means that the test case has already been executed and does not need to be executed again (hashing or other recording techniques can be applied to prevent re-execution of adaptive test cases). The meanings of the symbols are as follows:

- "·" means that the parameter's value is modified.
- "•" means that the parameter's value is the same with that in V_s.
- The parameters in square brackets "`[]`" are in U.
- The parameters in slashes "`//`" are *param*'s.
- The parameters in parentheses "`()`" are in *interaction*.
- The parameters in braces "`{}`" are in C_{tabu}.
- The parameters with no brackets or slashes indicate that their values keep the same with that in V_s.

Now, we explain how `FINOVLP` and `FIC` work on Example 2. Here we only choose the most significant runs of function *FINOVLP*, *FIC* and *LocateFixedParam* for explanation.

Table 2: Adaptive Test Cases Generated by FINOVLP+FIC

Test Cases								$Run(V1)$
1	2	2 p_1	1 p_2	2	2 p_1	1	1	fail
•	•	•	•	•	•	•	•	fail (SKIP)
/·/	•	•	•	•	•	•	•	fail
[·]	/·/	•	•	•	•	•	•	fail
[·]	[·]	/·/	•	•	•	•	•	fail
[·]	[·]	[·]	/·/	•	•	•	•	pass
[·]	[·]	[·]	(•)	/·/	•	•	•	fail
[·]	[·]	[·]	(•)	[·]	/·/	•	•	fail
[·]	[·]	[·]	(•)	[·]	[·]	/·/	•	fail
[·]	[·]	[·]	(•)	[·]	[·]	[·]	/·/	fail
•	•	•	{·}	•	•	•	•	fail
/·/	•	•	{·}	•	•	•	•	fail
[·]	/·/	•	{·}	•	•	•	•	fail
[·]	[·]	/·/	{·}	•	•	•	•	pass
[·]	[·]	(•)	{·}	/·/	•	•	•	fail
[·]	[·]	(•)	{·}	[·]	/·/	•	•	pass
[·]	[·]	(•)	{·}	[·]	(•)	/·/	•	fail
[·]	[·]	(•)	{·}	[·]	(•)	[·]	/·/	fail
•	•	{·}	{·}	•	{·}	•	•	pass

- *FINOVLP* is the top-level function of the algorithm, it locates non-overlapping faulty interactions of the seed test case. The input V_s = (1,2,2,1,2,2,1,1), S = (2,2,2,2,2,2,2,2) and $k = 8$. First the function initializes local variables: *interactions* $= \emptyset$, $C_{tabu} = \emptyset$. And then it enters the while-loop:
  - In the 1st cycle, *FINOVLP* first runs test case $\mathcal{M}(V_s, S, C_{tabu})$ = (1,2,2,1,2,2,1,1) (skipped) and the test fails. Then it calls *FIC* and locates a faulty interaction: *interaction* $= \{\hat{v}_4\}$. And then it updates local variables: *interactions* $= \{\{\hat{v}_4\}\}$ and $C_{tabu} = \{\hat{v}_4\}$.
  - In the 2nd cycle, the function first runs test case $\mathcal{M}(V_s, S, C_{tabu})$ = (1,2,2,2,2,2,1,1) and the test fails. Then it locates a faulty interaction: *interaction* $= \{\hat{v}_3, \hat{v}_6\}$. And then it updates local variables: *interactions* $= \{\{\hat{v}_4\}, \{\hat{v}_3, \hat{v}_6\}\}$ and $C_{tabu} = \{\hat{v}_3, \hat{v}_4, \hat{v}_6\}$.
  - In the 3rd cycle, the function runs a test case $\mathcal{M}(V_s, S, C_{tabu})$ = (1,2,1,2,2,1,1,1) and the test passes. Then the loop terminates

  At last, the function returns *interactions* $= \{\{\hat{v}_4\}, \{\hat{v}_3, \hat{v}_6\}\}$ as the result.
- Function *FIC* locates one faulty interaction of V_s containing only those parameters in $\{\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_k\} \backslash C_{tabu}$. Here we show how the function works in the second call which located faulty interaction $\{v_3, v_6\}$. The inputs V_s, S and k are the same with *FINOVLP*, $C_{tabu} = \{\hat{v}_4\}$. First, the function initializes local variables: *interaction* $= \emptyset$, $C_{free} = C_{tabu} = \{\hat{v}_4\}$. Then it enters the while-loop:
  - In the 1st cycle, *FIC* calls *LocateFixedParam* and locates a fixed parameter *param* $= \hat{v}_3$ and $C_{free} = \{\hat{v}_1, \hat{v}_2, \hat{v}_4\}$. Then the function updates local variable: *interaction* $= \{\hat{v}_3\}$.
  - In the 2nd cycle, the function locates a fixed parameter *param* $= \hat{v}_6$ and $C_{free} = \{\hat{v}_1, \hat{v}_2, \hat{v}_4, \hat{v}_5\}$. Then *interaction* $= \{\hat{v}_3, \hat{v}_6\}$.
  - In the 3rd cycle, the function locates no fixed parameter (*param* = null), and the while-loop terminates.

  After the while-loop terminates, the function returns *interaction* $= \{\hat{v}_3, \hat{v}_6\}$.
- Function *LocateFixedParam* locates one fixed parameter of $interaction_{new}$. Here we show how the function works in the call of *LocateFixedParam* which located fixed parameter $\hat{v}_6$. The inputs V_s, S and k are identical with *FIC*, $C_{free} = \{\hat{v}_1, \hat{v}_2, \hat{v}_4\}$, *interaction* $= \{\hat{v}_3\}$. First the function initializes the local variables: $C_{cand} = \{\hat{v}_5, \hat{v}_6, \hat{v}_7, \hat{v}_8\}$, $U = C_{free} = \{\hat{v}_1, \hat{v}_2, \hat{v}_4\}$. Then it enters the while-loop:
  - In the 1st cycle, the function extracts $\hat{v}_5$ from C_{cand}, so *param* $= \hat{v}_5$ and $C_{cand} = \{\hat{v}_6, \hat{v}_7, \hat{v}_8\}$. Then it runs an adaptive test case $\mathcal{M}(V_s, S, U \cup \{param\})$ = (2,1,2,2,1,2,1,1) and the test case fails. Then the function updates local variable: $U = \{\hat{v}_1, \hat{v}_2, \hat{v}_4, \hat{v}_5\}$.
  - In the 2nd cycle, the function extracts $\hat{v}_6$ from C_{cand}, so that *param* $= \hat{v}_6$ and $C_{cand} = \{\hat{v}_7, \hat{v}_8\}$. Then it runs an adaptive test case V =(2,1,2,2, 1,1,1,1) and the test case passes. So the function returns *param* $= \hat{v}_6$ and $U = \{\hat{v}_1, \hat{v}_2, \hat{v}_4, \hat{v}_5\}$.

3.3 FIC_BS: FIC with Binary Search

We will show in the later part of this paper that FIC uses at most k adaptive test cases to locate a faulty interaction. However, this number is relatively large as k grows. We know that *LocateFixedParam* locates a parameter $\hat{v}_i$ such that $\mathcal{M}(V_s, S, U)$ fails and $\mathcal{M}(V_s, S, U \cup \{\hat{v}_i\})$ passes for some U. Then we can apply binary search in this function to reduce the number of adaptive test cases to locate a faulty interaction. We call this alternative *faulty interaction characterization with binary search* (FIC_BS). The difference between FIC_BS from FIC is to call *BSLocateFixedParam* instead of *LocateFixedParam*. See the pseudo-code in Algorithm 4.

Table 3: Adaptive Test Cases Generated by FINOVLP+FIC_BS

Test Cases								$Run(V1)$
1	2	2 p_1	1 p_2	2	2 p_1	1	1	fail
•	•	•	•	•	•	•	•	fail (SKIP)
/·/	/·/	/·/	/·/	/·/	/·/	/·/	/·/	pass
/·/	/·/	/·/	/·/	•	•	•	•	pass
/·/	/·/	•	•	•	•	•	•	fail
[·]	[·]	/·/	•	•	•	•	•	fail
[·]	[·]	[·]	(•)	/·/	/·/	/·/	/·/	fail
•	•	•	{·}	•	•	•	•	fail
/·/	/·/	/·/	{·}	/·/	/·/	/·/	/·/	pass
/·/	/·/	/·/	{·}	/·/	•	•	•	pass
/·/	/·/	•	{·}	•	•	•	•	fail
[·]	[·]	/·/	{·}	•	•	•	•	pass
[·]	[·]	(•)	{·}	/·/	/·/	/·/	/·/	pass
[·]	[·]	(•)	{·}	/·/	/·/	•	•	pass
[·]	[·]	(•)	{·}	/·/	•	•	•	fail
[·]	[·]	(•)	{·}	[·]	(•)	/·/	/·/	fail
•	•	{·}	{·}	•	{·}	•	•	pass

Algorithm 4 Locate Fixed Parameter with Binary Search

```
1: function BSLocateFixedParam(V_s,S,k,C_free,interaction)
2:     C_cand ← {v̂_1, v̂_2, ..., v̂_k} \ C_free \ interaction
3:     U ← C_free
4:     if C_cand = ∅ or Run(M(V_s, S, U ∪ C_cand)) = fail then
5:         return (U, null)
6:     while |C_cand| > 1 do
7:         (C_low, C_high) ← Partition(C_cand)
8:         if Run(M(V_s, S, U ∪ C_low)) = pass then
9:             C_cand ← C_low
10:        else
11:            C_cand ← C_high
12:            U ← U ∪ C_low
13:    return (U, Extract(C_cand))
```

The function *Partition* at line 7 is used to partition C_{cand} into two parts: C_{low} and C_{high}, ensuring $|C_{low}| = \lceil \frac{|C_{cand}|}{2} \rceil$.

In function *BSLocateFixedParam*, the local variable C_{cand} is the set of all possible fixed parameters. U is the set of already-determined free parameters of the faulty interaction. The function first initializes local variables (line 2-3). Then it runs $\mathcal{M}(V_s, S, U \cup C_{cand})$ to determine whether there is a fixed parameter to locate (line 4-5). In the while-loop, the function first partitions C_{cand} into C_{low} and C_{high} (line 7). If $Run(\mathcal{M}(V_s, S, U \cup C_{low})) =$ `pass`, then there must be a fixed parameter in C_{low} (line 9). Otherwise, there must be a fixed parameter in C_{high}, and all parameters in C_{low} are free parameters (line 11-12). After the while-loop the function returns the only element in C_{cand} as the fixed parameter and U as the set of all already-determined free parameters (line 13).

Table 3 shows how `FIC_BS` works on Example 2. The meanings of the symbols are as follows:

- "·" means that the parameter's value is modified.
- "•" means that the parameter's value is the same with that in V_s.
- The parameters in square brackets "`[]`" are in U.
- The parameters in slashes "`//`" are in C_{cand} before entering the while-loop or in C_{low} in the while-loop.
- The parameters in parentheses "`()`" are in *interaction*.
- The parameters in braces "`{}`" are in C_{tabu}.
- The parameters with no brackets or slashes indicate that their values keep the same with that in V_s.

Function *BSLocateFixedParam* locates one fixed parameter of $interaction_{new}$. Here we show how the function works in the call of *BSLocateFixedParam* which located fixed parameter $\hat{v}_6$. The inputs V_s, S and k are the same with *FIC*, $C_{free} = \{\hat{v}_1, \hat{v}_2, \hat{v}_4\}$, $interaction = \{\hat{v}_3\}$. First the function initializes local variables: $C_{cand} = \{\hat{v}_5, \hat{v}_6, \hat{v}_7, \hat{v}_8\}$, $U = C_{free} = \{\hat{v}_1, \hat{v}_2, \hat{v}_4\}$. Because we have $C_{cand} \neq \emptyset$, the function runs $\mathcal{M}(V_s, S, \{\hat{v}_1, \hat{v}_2, \hat{v}_4, \hat{v}_5, \hat{v}_6, \hat{v}_7, \hat{v}_8\}) =$ `(2,1,2,2,1,1,2,2)` and the test case passes. So we know there is a fixed parameter in C_{cand}. Then the function enters the while-loop:

- In the 1st cycle, the function partitions C_{cand} into two sets: $C_{low} = \{\hat{v}_5, \hat{v}_6\}$ and $C_{high} = \{\hat{v}_7, \hat{v}_8\}$. Then it runs an adaptive test case $\mathcal{M}(V_s, S, U \cup C_{low}) =$ `(2,1,2,2,1,1,1,1)`. The test case passes so there is a fixed parameter in C_{low}. Let $C_{cand} = C_{low} = \{\hat{v}_5, \hat{v}_6\}$.
- In the 2nd cycle, the function partitions C_{cand} into two sets: $C_{low} = \{\hat{v}_5\}$ and $C_{high} = \{\hat{v}_6\}$. Then it runs an adaptive test case $\mathcal{M}(V_s, S, U \cup C_{low}) =$ `(2,1,2,2,1,2,1,1)`. The test case fails so there is a fixed parameter in C_{high} and $\hat{v}_5$ is a free parameter. Let $C_{cand} = C_{high} = \{\hat{v}_6\}$ and $U = \{\hat{v}_1, \hat{v}_2, \hat{v}_4, \hat{v}_5\}$.
- In the 3rd cycle, $|C_{cand}| = 1$ so the while-loop terminates.

After the while-loop terminates, $Extract(C_{cand}) = \hat{v}_6$ is returned as the fixed parameter and $U = \{\hat{v}_1, \hat{v}_2, \hat{v}_4, \hat{v}_5\}$ as set of all already-determined free parameters.

3.4 Complexity Analysis

Now we discuss the complexity of our three algorithms: `FIC`, `FIC_BS` and `FINOVLP`. The extra cost of running the three algorithms is very low compared with the execution cost of SUT (i.e. the cost of function *Run*). So we only analyze the number of adaptive test cases.

- For `FIC`, we can see that $|C_{cand}|$ in *LocateFixedParam* decreases by 1 for each execution of an adaptive test

case. So the total number of adaptive test cases to locate one faulty interaction is no more than k because $|C_{cand}| \leq k$ in the first call of *LocateFixedParam* when locating this faulty interaction.

- For `FIC_BS`, we can see that the number of adaptive test cases to locate one fixed parameter is equal to or less than $\lceil \log_2 k \rceil + 1$. And we can know that in the last call of *BSLocateFixedParam*, no fixed parameters are located and only one adaptive test case is executed. Suppose the size of the faulty interaction is t, then the total number of adaptive test cases for the faulty interaction is no more than $t(\lceil \log_2 k \rceil + 1) + 1$.

- For `FINOVLP`, we can see that exactly one test case will be executed for each located faulty interaction. So if the algorithm has located d faulty interactions, the total number of tests executed at line 5 is $d + 1$.

4. ABOUT THE ASSUMPTIONS

In this section, we discuss the assumptions we made in Section 3.

4.1 Weakening the Assumptions

In the Appendix A, we prove the correctness of our `FIC` and `FIC_BS` under the four assumptions we made in Section 3.1. Although many studies on faulty interaction characterization have made the same assumptions explicitly or implicitly as we did, those assumptions do not always hold in many practical applications. So we have to discuss how to weaken the assumptions. Up to now, we have some empirical solutions. However, the effectiveness of these solutions still needs further studies.

4.1.1 Assumption 1

In some applications, the SUT fails with multiple failure types. A solution is to use the appearance of a specific failure type instead of *pass* and $fail$. For instance, suppose we know V_s fails with types `err1` and `err2`, and we are characterizing the faulty interactions for `err1`, then we consider a test passes when `err1` does not appear in the failure type.

The Delta Debugging algorithms[18] used `unresolved` outcome when we cannot determine whether some specific failure occurs or not. Here we label the `unresolved` tests with `pass`. The effect of this change still needs further studies.

4.1.2 Assumption 2

If this assumption does not hold and a test case matching one or more faulty interactions passes by coincidence, then several irrelevant parameters will be decided as fixed parameters of the faulty interaction under locating, so that `FIC` and `FIC_BS` can locate a super-interaction of the real faulty interaction. This super-interaction is still capable of helping the developer to fix the bug.

4.1.3 Assumption 3

Many applications have dependent parameters. A solution is to label all invalid test cases with `pass`. Similar with weakening Assumption 2, several irrelevant parameters will be decided as fixed parameters. So `FIC` and `FIC_BS` can locate a super-interaction of the real faulty interaction in this condition. But this will increase the number of adaptive test cases for `FIC_BS`.

4.1.4 Assumption 4

Some faulty interaction characterization methods use a concept called *safe values* [8, 9]. A safe value for parameter $\hat{v}_i$ is a value $s_{safe,i} \in \{1, 2, \ldots, s_i\}$ such that no faulty interaction of the SUT contains the fixed parameter $\hat{v}_i$ with $v_i = s_{safe,i}$.

If safe values exist, we can use them to modify parameter values in V_s. This ensures that Assumption 4 holds when we apply `FIC` in practical software. On the other hand, the parameters with safe values can be directly placed in the tabu list, so the number of adaptive test cases can be reduced. However, safe values do not always exist, so new faulty interactions may appear when we modify parameters of V_s. Then `FIC` and `FIC_BS` may locate some incorrect faulty interactions. This seems to be disappointing. But in fact, the fixed parameters located by `FIC` are still fixed parameters of some real faulty interactions of the SUT. And moreover, in this condition we may have detected new faults in the SUT, and it can be revealed during regression testing. This may help increase the reliability of the software.

4.2 Threats to Validity

We have discussed in Section 4.1 on how to weaken the four assumptions. One threat of `FIC` is brought by Assumption 4. We mentioned that if Assumption 4 does not hold, `FIC` and `FIC_BS` will locate some incorrect faulty interactions or even cannot locate faulty interactions. We cannot expect that safe values always exist to satisfy Assumption 4. So it is better to apply `FIC` and `FIC_BS` when the software contains not too many faults, so that new faulty interactions are less likely to appear when modifying parameters of V_s. By the way, when the software contains too many faults, it needs human re-examination rather than just doing automatic diagnosis.

Another threat is brought by Assumption 3. When the input parameters have too many constraints, `FIC_BS` may get too many invalid tests, which will be labeled as `pass`. Then many free parameters will be determined as fixed parameters. From the complexity of `FIC_BS`, we can know the number of adaptive tests grows linearly with the size of the located faulty interaction. So if there are too many constraints on input parameters, the adaptive tests needed by `FIC_BS` may increase.

However, these are not problems only for `FIC` and `FIC_BS`. Almost all adaptive methods will face the same problems.

5. RELATED WORK

Previous works in faulty interaction characterization can be categorized into two kinds of approaches:

The first kind of approach is to do post-analysis of the test results of CAs. This includes the work of Yilmaz et al. [17]. They applied classification tree technique to analyze the execution result of CAs. The main idea is to find the differences between passing and failing tests. However, CA is not adequate for precise faulty interaction characterization. A weakness of post-analysis methods is that with large size of faulty interactions, large number of failing tests, and small size of the test suite, there is little information to exclude the good interactions. For instance, assume the test suite and the test result are as shown in Table 1, these methods will not work.

Table 4: Comparison of Methods for Locating Faulty Interactions

	Limitations			Results	
Method	t	d	s	Faulty Interactions Located	Number of Test Cases
Martínez's 1st Adaptive [8, 9]	no	no	no	$t \leq 2$	$O(\hat{d} \log k + \hat{d}^2)$
Martínez's 2nd Adaptive [8, 9]	no	no	$s = 2$	$t \leq 2$	expected $O(d^2 + d \log k + \log^c k)$
Martínez's ELA [8, 9]	given	given	no	all	$O(ds^d \log k)$ (for fixed t)
Shi's `AIFL` [13, 16]	no	$d \leq 1$	no	all ($d \leq 1$)	k
Wang's `IterAIFL` [15]	non-overlapping			all	not studied
Zeller's `ddmin` [18]	no	no	no	depends on the upper level	$\log_2 k \sim k^2 + 3k$/interaction
`FIC`	no	no	no	depends on the upper level	$\leq k$/interaction
`FIC_BS`	no	no	no	depends on the upper level	$\leq t(\lceil \log_2 k \rceil + 1) + 1/t$-way interaction

The second kind of approach is to design new test generation techniques to characterize the faulty interactions, which can be classified into two categories:

- Non-adaptive methods, where test cases are independent and can be executed in parallel.
- Adaptive methods, which use the execution results of previous tests to generate new test cases.

Table 4 shows a comparison of `FIC` and `FIC_BS` with some methods in the test generation approach.

The non-adaptive methods include *locating and detecting arrays* (LDAs) proposed by Colbourn et al. [3] and *error locating arrays* (ELAs) proposed by Martínez et al. [8, 9]. Generating an LDA or ELA requires the number of parameters and their possible values, a given strength t and a maximum number of faulty interactions d. After running the LDA or ELA as the test suite, we can detect and locate at most d faulty interactions of size equal to or less than t. However, the generation of small LDAs and ELAs is still a challenging problem, and the size of the test suite is very large.

The adaptive methods include Zeller's Delta Debugging [18], Shi's `AIFL` method [13, 16], Wang's `IterAIFL` method [15], and two adaptive algorithms proposed by Martínez et al. [8, 9]. `FIC` and `FIC_BS` belong to this category too. What the adaptive methods have in common is that test cases are generated by modifying parameter values of the failing test cases, and most of them have explicitly or implicitly used some of the assumptions we made in Section 3.1.

Delta Debugging is very effective and has been used to locate failure causes in some real systems. It has two algorithms: `ddmin` and `dd`. The former one is used to locate one faulty interaction in V_s, and the latter one is used to isolate a part of the V_s which is relevant to the failure. All of the two algorithms are divide-and-conquer approaches. When there are two test outcomes (`pass` and `fail`), `FIC`, `FIC_BS` and `ddmin` have the same functionality. Algorithm 2, Algorithm 4 and `dd` have the same functionality too. And `dd` is in fact reduced to Algorithm 4. One drawback of `ddmin` is that the number of adaptive tests is unstable as it is sensitive to the order of parameters.

Shi's `AIFL` method needs k adaptive test cases. The adaptive test suite consists of all tests that have modified exactly one parameter of V_s. The weakness of `AIFL` is that it does not work in case that there are multiple faulty interactions in the seed test case.

Wang's `IterAIFL` method is an extension to Shi's method, which can deal with multiple faulty interactions. They process all failed tests together, and generate test cases to reduce and minimize the set of suspicious faulty interactions. `IterAIFL` use *mutation strength* to control the number of parameters to be modified. In each iteration, the mutation strength t_m is increased, and the newly-generated test cases are all that have modified precisely t_m parameters of failing tests. `IterAIFL` requires the faulty interactions to be non-overlapping, and the number of adaptive test cases is not studied.

The two adaptive algorithms of Martínez have no limits on the size or the number of faulty interactions in the seed test case. However they can only locate faulty interactions of no more than 2 parameters. The 2nd algorithm does not assume safe values exist, but works only when $s = 2$ for all parameters. In Table 4, the "c" in the number of test cases of the 2nd algorithm is an upper bound, which satisfies $d \leq \frac{c}{2} \log \log k$.

Compared with the above methods, our methods can characterize non-overlapping faulty interactions of all sizes. `FIC` and `FIC_BS` need fewer adaptive test cases (shown in Section 6.1), and have equivalent or weaker restrictions. These two methods have no limits on the size or the number of faulty interactions. Under our four assumptions made in Section 3.1, the ability to locate faulty interactions of our methods is equivalent to `ddmin`, while is stronger than the other methods mentioned above. For `FIC`, the number of adaptive test cases to locate one faulty interaction is k. And for `FIC_BS`, the number is $t(\lceil \log_2 k \rceil + 1) + 1$, which is an exciting result. Same with `ddmin`, the faulty interactions located by `FIC` and `FIC_BS` depend on the upper-level implementation. (e.g. We can use `FINOVLP` to locate non-overlapping faulty interactions.)

For dealing with parameter constraints, Hierarchical Delta Debugging [10] runs on an upper level of `ddmin`, and can effectively reduce the number of adaptive tests for minimizing structured inputs such as HTML or XML files. Our methods can easily fit for this technique.

6. EXPERIMENTAL RESULTS

6.1 Simulation Results

For setting up the experiment, we design a toy SUT containing some defects. We set $s = 2$ for simplicity, and all parameters have safe values. We think that `ddmin` algorithm is the most popular and works best among all existing methods, so we are comparing `FIC` and `FIC_BS` with `ddmin`.

We assume that the SUT has exactly one randomly generated faulty interaction of fixed size t. Then we vary the size of the faulty interaction t and the number of parameters k to compare the number of adaptive test cases needed by the three algorithms. The results are shown in Table 5. From

Table 5: Number of Adaptive Test Cases (a)

t	k	ddmin	FIC	FIC_BS
1	4	3.1	4	3.76
	8	4.58	8	4.82
	16	5.79	16	5.98
	32	7.45	32	6.98
	64	9.00	64	7.98
	128	10.49	128	8.99
	256	11.99	256	9.99
2	4	7.19	4	5.64
	8	12.63	8	8.09
	16	19.16	16	10.17
	32	25.79	32	12.35
	64	31.93	64	14.45
	128	37.86	128	16.37
	256	45.79	256	18.42
3	4	10.59	4	7.59
	8	22.85	8	11.25
	16	32.76	16	14.23
	32	46.32	32	17.26
	64	61.61	64	20.45
	128	76.61	128	23.50
	256	91.40	256	26.34
4	4	10.00	4	9.00
	8	26.04	8	13.67
	16	44.62	16	18.17
	32	64.50	32	22.04
	64	82.38	64	26.17
	128	105.61	128	30.04
	256	126.74	256	34.23
5	4	-	-	-
	8	28.91	8	16.05
	16	52.35	16	21.69
	32	78.12	32	26.75
	64	105.67	64	32.16
	128	134.87	128	36.88
	256	164.97	256	42.09
6	4	-	-	-
	8	30.23	8	18.69
	16	58.83	16	25.37
	32	92.98	32	31.74
	64	129.18	64	38.30
	128	165.63	128	44.39
	256	205.42	256	50.25

Table 6: Number of Adaptive Test Cases (b)

t/k	k	ddmin	FIC	FIC_BS
0.125	8	4.44	8	4.88
	16	19.37	16	10.34
	32	64.72	32	22.28
	64	171.28	64	48.98
	128	481.24	128	110.22
	256	1510.41	256	246.53
0.250	8	12.15	8	7.83
	16	45.40	16	18.01
	32	118.48	32	40.87
	64	321.17	64	92.56
	128	999.39	128	221.80
	256	3428.29	256	483.25
0.375	8	21.58	8	10.80
	16	57.83	16	25.08
	32	156.63	32	57.58
	64	457.81	64	136.34
	128	1456.23	128	312.89
	256	5149.38	256	713.71
0.500	8	27.24	8	13.74
	16	71.22	16	32.50
	32	182.56	32	75.02
	64	553.32	64	177.97
	128	1798.89	128	411.64
	256	6315.79	256	942.39
0.625	8	28.93	8	16.12
	16	77.41	16	39.04
	32	200.80	32	92.63
	64	588.23	64	217.85
	128	1909.40	128	506.77
	256	6598.10	256	1164.97
0.750	8	29.45	8	18.53
	16	77.08	16	45.32
	32	202.61	32	108.63
	64	569.44	64	256.92
	128	1767.65	128	600.93
	256	6090.68	256	1382.96
0.875	8	28.65	8	20.87
	16	71.56	16	51.43
	32	171.38	32	123.65
	64	457.15	64	295.68
	128	1327.45	128	692.61
	256	4178.97	256	1596.01

the results we can see that in most of the cases where $t \leq 6$, FIC or FIC_BS needs fewer adaptive test cases than ddmin.

Then, we want to see how the three algorithms perform for locating faulty interactions in large sizes. So we design the following experiment: assume that the SUT has exactly one randomly generated faulty interaction of size t. We vary t/k and k, and compare the number of adaptive test cases needed by the three algorithms. The results are shown in Table 6. We can see that our methods perform better in most of the cases. However, the results of random experiments do not mean ddmin performs worse for faulty interactions of large sizes. The number of adaptive test cases needed by ddmin is sensitive to the order of parameters, it may work better when dealing with failures caused by long consecutive inputs (e.g. long XML tags).

6.2 Experiments with Real Programs

We have applied FIC on the six programs (**count**, **tokens**, **series**, **nametbl**, **ntree** and **cmdline**) provided in [7] to locate the faulty interactions in them. Each program has some bugs. We performed combinatorial testing on the test subjects and found several failing test cases. Due to different modeling of the test subjects, our result may be different from previous work using the same test subjects (such as [5] and [4]). Because we are showing whether FIC can locate faulty interactions for failures detected in combinatorial testing, we do not consider too much about fault coverage.

Then we applied our methods and located the faulty interactions. Each failing test case $V_{s,i}$ is processed as follows:

- For each failure type of $V_{s,i}$, check if $V_{s,i}$ matches any located faulty interactions of this type.
- If $V_{s,i}$ matches no faulty interactions of some type, do FIC to locate a new faulty interaction for this type.

By doing the procedures above, we located faulty interactions for all the failures detected in combinatorial testing.

Table 7 shows a summary of our results on the six test subjects, and Figure 1 shows the distribution of faulty interactions of different sizes.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a new automated fault characterization method FIC and its binary search alternative

Table 7: Models of the Six Experiment Subjects

Program	Model	Fault Types	# Original Tests	# Adaptive Tests	# Faulty Interactions
count	$SUT t(6; 2^2 3^4)$	5	14	14	10
series	$SUT(4; 2^1 4^2 6^1)$	4	26	10	5
tokens	$SUT(4; 2^2 3^2)$	3	12	7	5
ntree	$SUT(6; 2^2 4^4)$	4	16	27	13
nametbl	$SUT(8; 2^4 3^2 5^2)$	7	25	65	22
cmdline	$SUT(9; 2^1 3^4 4^1 6^2 15^1)$	4	38	0	10

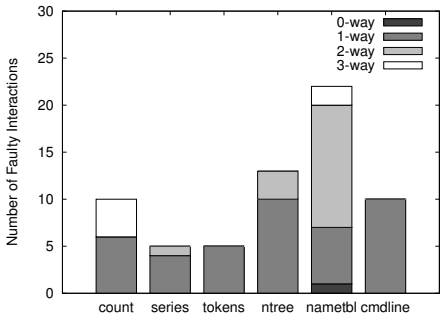

Figure 1: Faulty Interactions of Different Sizes

`FIC_BS` based on adaptive black-box testing. And we provide a tradeoff strategy of locating multiple non-overlapping faulty interactions. For locating one faulty interaction of size t in an SUT having k parameters, `FIC` runs at most k adaptive test cases, and `FIC_BS` runs at most $t(\lceil \log_2 k \rceil + 1) + 1$ adaptive test cases. Our simulation results show that `FIC` and `FIC_BS` work better than `ddmin` in most cases of locating randomly-generated faulty interactions. While in practical applications, knowledge of the SUT may be used to help decide which algorithm to use.

The features of `FIC` and `FIC_BS` are very exciting and they are ready to be applied in practical applications. This inspires our future work to increase the performance and applicability of our method:

- We used some common assumptions of the faulty interaction characterization algorithms to help design `FIC` and `FIC_BS`. As those assumptions do not always hold, we proposed some empirical solutions of weakening those assumptions. But we should find more effective solutions to work on real software systems, such as dealing with the absence of safe values, faulty interactions of large sizes and parameter constraints.
- Another work to do is to find a way to make full use of these faulty interactions. Only locating faulty interactions is not enough for a complete debugging process. It requires other techniques to locate bugs in the software. Combining our methods and other fault locating techniques is another work to do in the future.

Acknowledgements

We are grateful to Jun Yan, Zhenyu Zhang and Changhai Nie for their helpful comments and suggestions.

8. REFERENCES

[1] D. Cohen, S. Dalal, M. Fredman, and G. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[2] M. Cohen, P. Gibbons, W. Mugridge, and C. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 38–48. IEEE.

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

[4] M. Grindal, B. Lindström, J. Offutt, and S. Andler. An evaluation of combination strategies for test case selection. *Empirical Software Engineering*, 11(4):583–611, 2006.

[5] E. Kamsties and C. Lott. An Empirical Evaluation of Three Defect-Detection Techniques. In *Proceedings of the 5th European Software Engineering Conference (ESEC 1995)*, pages 362–383. Springer.

[6] D. Kuhn and M. Reilly. An Investigation of the Applicability of Design of Experiments to Software Testing. In *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW 2002)*, pages 91–95. IEEE.

[7] C. Lott. A repeatable software engineering experiment. `http://www.maultech.com/chrislott/work/exp/`.

[8] C. Martínez, L. Moura, D. Panario, and B. Stevens. Algorithms to locate errors using covering arrays. In *LATIN 2008: Theoretical Informatics*, volume 4957 of *Lecture Notes in Computer Science*, pages 504–519. Springer.

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

[10] G. Misherghi and Z. Su. HDD: hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pages 142–151. ACM.

[11] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43:11:1–11:29, 2011.

[12] C. Nie, H. Leung, and B. Xu. The minimal failure-causing schema of combinatorial testing. *ACM Transactions on Software Engineering and Methodology*, 2011, to appear.

[13] L. Shi, C. Nie, and B. Xu. A software debugging method based on pairwise testing. In *Proceedings of the International Conference on Computational Science (ICCS 2005)*, volume 3516 of *Lecture Notes in Computer Science*, pages 55–81. Springer.

[14] D. Torney. Sets pooling designs. *Annals of Combinatorics*, 3(1):95–101, 1999.

[15] Z. Wang, B. Xu, L. Chen, and L. Xu. Adaptive Interaction Fault Location Based on Combinatorial Testing. In *Proceedings of the 10th International Conference on Quality Software (QSIC 2010)*, pages 495–502. IEEE.

[16] B. Xu, C. Nie, L. Shi, and H. Chen. A software failure debugging method based on combinatorial design approach for testing. *Jisuanji Xuebao(Chinese Journal of Computers)*, 29(1):132–138, 2006.

[17] C. Yilmaz, M. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *IEEE Transactions on Software Engineering*, pages 20–34, 2006.

[18] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, pages 183–200, 2002.

APPENDIX

A. PROOF OF THE CORRECTNESS OF FIC

It is a question whether the faulty interactions located by `FIC` and `FIC_BS` are exact faulty interactions of V_s. Here we give a proof for the correctness of `FIC` and `FIC_BS`.

We prove that each faulty interaction returned by function *FIC* is a faulty interaction of V_s. Denote the variable *interaction* returned by *FIC* as $interaction^*$. Then we have:

COROLLARY 1. *All faulty interactions of* $\mathcal{M}(V_s, S, \{\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_k\} \setminus interaction^*)$ *are sub-interactions of* $interaction^*$.

PROOF. This is a special case of Lemma 1 when $U = \{\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_k\} \setminus interaction^*$. □

After the i^{th} ($0 \leq i \leq t+1$, t is the size of $interaction^*$) iteration of the while-loop in function *FIC*, we denote the variable *interaction* as $interaction_i$, the variable *param* as $param_i$, the variable C_{free} as $C_{free,i}$. It is easy to know that $interaction_t = interaction_{t+1} = interaction^*$.

LEMMA 4. *For all* $0 \leq i \leq t+1$, $\mathcal{M}(V_s, S, C_{free,i})$ *must fail, and all faulty interactions of it are super-interactions of* $interaction_i$.

PROOF. We proceed by induction on the number of iterations:

1. $i = 0$. We know $interaction_0 = \emptyset$, $C_{free,i} = C_{tabu}$ and $\mathcal{M}(V_s, S, C_{tabu})$ fails so Lemma 4 holds.
2. We assume that Lemma 4 holds when $i = m$. We prove that it holds when $i = m+1$.
   - If no fixed parameter is located in the $(m+1)^{\text{th}}$ cycle, then we know that $C_{free,m+1} \supseteq C_{free,m}$ and $interaction_{m+1} = interaction_m$. Applying Lemma 2, we know that all faulty interactions of $\mathcal{M}(V_s, S, C_{free,m+1})$ are of $\mathcal{M}(V_s, S, C_{free,m})$, thus are super-interactions of $interaction_{m+1}$. In addition, $\mathcal{M}(V_s, S, C_{free,m+1})$ must fail, or else a new fixed parameter will be located. So Lemma 4 holds in this condition.
   - If a fixed parameter $\hat{v}_j$ is located in the $(m+1)^{\text{th}}$ cycle, then we know that $\mathcal{M}(V_s, S, C_{free,m+1})$ will fail and $\mathcal{M}(V_s, S, C_{free,m+1} \cup \{\hat{v}_j\})$ will pass, where $C_{free,m+1} \cup interaction_m = \emptyset$ and $C_{free,m+1} \supseteq C_{free,m}$.
     a) By applying Lemma 3, we know that $\hat{v}_j$ must be a fixed parameter of all faulty interactions of $\mathcal{M}(V_s, S, C_{free,m+1})$.
     b) From induction hypothesis and Lemma 2 we know all faulty interactions of $\mathcal{M}(V_s, S, C_{free,m+1})$ are super-interactions of $interaction_m$.
     c) From the function *FIC*, it is easy to see that $interaction_{m+1} = interaction_m \cup \{\hat{v}_j\}$.
     From a), b) and c), we know all faulty interactions of $\mathcal{M}(V_s, S, C_{free,m+1})$ must be super-interactions of $interaction_{m+1}$. In addition, we can know from the algorithm that $\mathcal{M}(V_s, S, C_{free,m+1})$ must fail, or else $\hat{v}_j$ will not be located as a fixed parameter. So Lemma 4 holds when $i = m+1$.

From 1) and 2), we know that Lemma 4 holds for all $i \geq 0$. □

COROLLARY 2. $\mathcal{M}(V_s, S, \{\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_k\} \setminus interaction^*)$ *must fail, and all faulty interactions of this test case are super-interactions of* $interaction^*$.

PROOF. Special case of Lemma 4 when $i = t+1$, as $interaction_{t+1} = interaction^*$. □

THEOREM 1. $interaction^*$ *is a faulty interaction of* V_s.

PROOF. From Corollary 1 and Corollary 2, we know all faulty interactions of $\mathcal{M}(V_s, S, \{\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_k\} \setminus interaction^*)$ equal to $interaction^*$.

From Corollary 2, we can see $\mathcal{M}(V_s, S, \{\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_k\} \setminus interaction^*)$ must fail. So we know that it has a faulty interaction, and thus $interaction^*$ must be a faulty interaction of it, and moreover, of V_s (Lemma 2). □