Identifying Failure-Inducing Combinations in a Combinatorial Test Set

Laleh Shikh Gholamhossein Ghandehari[1], Yu Lei[1], Tao Xie[2], Richard Kuhn[3], Raghu Kacker[3]

[1]Dept. of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019, USA
[2]Dept. of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
[3]Information Technology Lab, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA

***Abstract*- A t-way combinatorial test set is designed to detect failures that are triggered by combinations involving no more than t parameters. Assume that we have executed a t-way test set and some tests have failed. A natural question to ask is: what combinations have caused these failures? Identifying such combinations can facilitate the debugging effort, e.g., by reducing the scope of the code that needs to be inspected.**

In this paper, we present an approach to identifying failure-inducing combinations, i.e., combinations that have caused some tests to fail. Given a t-way test set, our approach first identifies and ranks a set of suspicious combinations, which are candidates that are likely to be failure-inducing combinations. Next, it generates a set of new tests, which can be executed to refine the ranking of suspicious combinations in the next iteration. This process can be repeated until a stopping condition is satisfied. We conducted an experiment in which our approach was applied to several benchmark programs. The experimental results show that our approach can effectively and efficiently identify failure-inducing combinations in these programs.

Keywords- Combinatorial Testing, Fault Localization, Debugging.

I. INTRODUCTION

Combinatorial testing has been shown to be a very practical and efficient testing strategy [2, 3, 6]. The main idea behind combinatorial testing is the following: while the behavior of a system as a whole may be affected by many parameters, many failures are caused by interactions of only a few parameters [5]. It is, however, not known *a priori* interactions of which parameters could cause a failure. A t-way combinatorial test set is designed to cover all the t-way interactions, i.e., combinations of values involving t parameters, where t is typically a small integer [2, 6]. If the input parameters are modeled properly, a t-way test set is guaranteed to detect all the failures that are triggered by interactions of no more than t parameters.

Assume that we have executed a t-way test set and some tests have failed. A natural question to ask is: what combinations have caused these failures? Identifying such combinations can facilitate the debugging effort, e.g., by reducing the scope of the code that needs to be inspected.

In this paper, we present an approach to identifying failure-inducing combinations in a combinatorial test set. A failure-inducing combination, or simply an inducing combination, is a combination of parameter values such that all test cases containing this combination fail [8, 10, 13]. Our approach takes as input a combinatorial test set and produces as output a ranking of t-way suspicious combinations in terms of their likelihood to be inducing. Moreover, our approach identifies all the suspicious combinations whose size is smaller than t, if they exist.

Our approach adopts an iterative framework. At each iteration, a set F of test cases is analyzed. (F is a t-way test set at the first iteration.) Our approach first identifies the set π of all t-way suspicious combinations, and then ranks them based on their likelihood to be inducing. Next, our approach generates a set F' of new test cases. The test cases in F', if executed, will be added to F, and will be analyzed in the next iteration to refine the set of suspicious combinations and their ranking. This process is repeated until a stopping condition is satisfied.

The novelty of our approach lies in the fact that we rank suspicious combinations based on two notions: suspiciousness of a combination and suspiciousness of the environment of a combination. Informally, the environment of a combination consists of other parameter values that appear in the same test case. The higher the suspiciousness of a combination, the lower the suspiciousness of its environment, the higher this combination is ranked. Moreover, new test cases are generated for the most suspicious combinations. Let f be a new test case generated for a suspicious combination c. Test f is generated such that the suspiciousness of the environment for c is minimized. If f fails, it is more likely to be caused by c instead of other values in f.

We report an experiment in which we apply our approach to a set of six third-party benchmark programs. Each benchmark program has a number of seeded faults. The results show that our approach is effective in identifying inducing combinations. On one hand, truly inducing combinations are ranked to the top very quickly. On the other hand, combinations that are ranked on the top but are not failure-inducing often have a very high probability to be inducing. Our approach is also very efficient in that only a very small percentage of all possible test cases need to be executed. For example, for one version of the six benchmark program (version 3 of a program named *cmdline*), the only

two inducing combinations are ranked to the top 10 after executing 0.034% of all possible test cases.

The remainder of this paper is organized as follows. Section II represents the definitions and notations used in this paper. Section III describes our approach. Section IV gives an example to illustrate our approach. Section V reports an experiment that demonstrates the effectiveness and efficiency of our approach. Section VI discusses existing work on identifying inducing combinations. Section VII provides some concluding remarks.

II. PRELIMINARIES

In this section, we introduce the basic definitions and assumptions needed in our approach.

A. Basic concepts

Assume that the system under test (SUT) has k input parameters, denoted by set $P = \{p_1, p_2, ..., p_k\}$. Let d_i be the domain of parameter p_i. That is, d_i contains all possible values that p_i could take, and let $D = \{d_1 \cup d_2 \cup ... \cup d_k\}$.

Definition 1. (*Test Case*) A test case is a function that assigns a value to each parameter. Formally, a test case is a function $f: P \rightarrow D$.

We use Γ to represent all possible test cases for the SUT. It is clear that $|\Gamma| = |d_1| \times |d_2| \times ... \times |d_K|$.

Definition 2. (*Test Oracle*) A test oracle determines whether the execution of a test case is "pass" or "fail". Formally, a test oracle is a function $r: \Gamma \rightarrow \{pass, fail\}$.

Definition 3. (*Combination*) A combination c is a test case f restricted to a non-empty, proper subset M of parameters in P. Formally, $c = f|_M$, where $M \subset P$, and $|M| > 0$.

In the preceding definition, M is a proper subset of P thus a test case is not considered to be a combination in this paper. We use $dom(c)$ to denote the domain of c, which is a set of parameters involved in c. (Note that $dom(c)$ is the domain of a function, which is different from the domain of a parameter.) We define the size $[c]$ of a combination c to be the number of parameters involved in c. That is $[c] = |dom(c)|$.

A combination of size 1 is a special combination, which we refer to as a component. Since there is only one parameter involved, we denote a component o as an assignment, i.e., $o = p \leftarrow v$, where $o(p) = v$.

Definition 4. (*Component Containment)* A component $o = p \leftarrow v$ is contained in a combination c denoted by $o \in c$, if and only if $p \in dom(c)$ and $c(p) = v$.

Definition 5. (*Combination Containment*) A combination c is contained in a test case f, denoted by $c \subset f$, if and only if $\forall p \in dom(c), f(p) = c(p)$.

Definition 6. (*Inducing Combination*) A combination c is failure-inducing if any test case f in which c is contained fails. Formally, $\forall f \in \Gamma: c \subset f \Longrightarrow r(f) = fail$.

Definition 6 is consistent with the definition of inducing combinations in previous work [8, 9, 10, 13].

Definition 7. (*Inducing Probability*) The inducing probability of a combination c is the ratio of the number of all possible failed test cases containing c to the number of all possible test cases containing c. The inducing probability is computed by

$$\frac{|\{f \in \Gamma | r(f) = fail \wedge c \subset f\}|}{|\{f \in \Gamma | c \subset f\}|}$$

The computation of inducing probabilities requires all possible test cases containing a combination; such represent is not possible in practice. This notion is mainly used to evaluate the goodness of our experimental results.

Definition 8. (*Suspicious Combination*) A combination c is a suspicious combination in a test set $F \subseteq \Gamma$ if c is contained only in failed test cases in F. Formally, $\forall f \in F: c \subset f \Rightarrow r(f) = fail$.

Inducing combinations must be suspicious combinations, but suspicious combinations may or may not be inducing combinations.

B. Assumption

Assumption 1. The output of the SUT is deterministic.

In other words, the SUT always produces the same output from a given test case.

Assumption 2. There exists a test oracle that determines the status of a test execution, i.e., "pass" or "fail".

Assumption 2 is made to simplify the presentation of our approach. The construction of a test oracle is an independent research problem. When a test oracle exists, our approach can be fully automated. When a test oracle does not exist, our approach can still be applied, but the user needs to assist in determining the execution status of a test case.

Assumption 3. Inducing combinations should involve no more than t parameters, where t is the strength of the initial combinatorial test set.

Our approach focuses on detecting inducing combinations that are of size t or less. Such focus is consistent with the implicit assumption held when a tester decides to use a t-way combinatorial test set.

III. APPROACH

In this section, we present our approach to identifying inducing combinations.

A. Framework

As shown in Fig. 1, the framework consists of three main steps. (1) *Rank generation*: In this step, we first identify all the t-way suspicious combinations in F (line 4). We then produce a ranking of the suspicious combinations (line 7). (2) *Test generation*: In this step, we generate a set of new tests, which will be used to refine the ranking of suspicious combinations in the next iteration (line 9). (3) *Reduction*: In this step, we analyze the final ranking of t-way suspicious combinations to derive suspicious combinations of size smaller than t, if they exist (line 17). The details of these three steps are presented in the following subsections.

In the framework, the two steps, rank generation and test generation, are performed iteratively when the set of suspicious combinations, π, is not empty and the size of π in the current iteration is less than the previous iteration (line 5). Otherwise, the algorithm stops (lines 12, 14).

In addition, another stopping condition happens when a combination is marked as an inducing combination by the test generation step (line 15). The reduction step analyzes π to determine smaller suspicious combinations and produce a ranking for them (line 17).

The user can decide to stop at the end of each iteration, if the resource is limited.

B. Rank generation

In step of rank generation, we first identify the set π' of all t-way suspicious combinations in F. In the first iteration, F is the initial t-way test set, i.e., F_0. Thus, F_0 covers all t-way combinations. Initially, π' contains all the t -way combinations. We then check each t-way combination c in π'. If c appears in at least one passed test, c is removed from π'. In the subsequent iterations, we do not have to re-compute π' from the scratch. Instead, we only need to remove from π' all the combinations contained in newly added, passed tests.

We next discuss how to rank the suspicious combinations in π. First, we introduce three important metrics of suspiciousness, suspiciousness of component, suspiciousness of combination, and suspiciousness of environment.

Suspiciousness of component (ρ): This notion is defined such that the higher ρ a component o has, the more likely o contributes to a failure, and the more likely o appears in an inducing combination. Let F be the test set that is analyzed in the current iteration. In our approach, ρ is computed by the following formula:

$$\rho(o) = \frac{1}{3}\big(u(o) + v(o) + w(o)\big) \qquad (1)$$

Where

$$u(o) = \frac{|\{f \in F_i | r(f) = \text{fail} \wedge o \in f\}|}{|\{f \in F_i | r(f) = \text{fail}\}|}$$

$$v(o) = \frac{|\{f \in F_i | r(f) = \text{fail} \wedge o \in f\}|}{|\{f \in F_i | o \in f\}|}$$

$$w(o) = \frac{|\{c | o \in c \wedge c \in \pi\}|}{|\pi|}$$

The first factor of (1), $u(o)$, shows the ratio of the number of failed test cases in which component o appears over the total number of failed test cases. The second factor, $v(o)$, shows the ratio of the number of failed test cases in which component o appears over the total number of test cases in which component o appears. The third factor shows the ratio of the number of suspicious combinations in which component o appears over the total number of suspicious combinations. The three factors are averaged to produce a value between 0 and 1.

The motivation behind the first two factors is that the more frequently a component appears in failed test cases, this component is more likely to contribute to a failure.

There is an important difference between the two factors. Since the greater the domain size is, the less frequently each individual value of this parameter appears in a test set and consequently in failed test cases, the first factor, $u(o)$, has a bias towards smaller domain size parameters. The second factor, $v(o)$, is brought in to reduce this bias.

Algorithm IdentifyInducingCombinations
Input: sut, F_0, t
Output: a set $R = \{R_1, R_2, \ldots R_t\}$ of rankings,
where R_i is the ranking of i-way suspicious combinations

```
1.  let F = F0 and let π be an empty set
2.  while (true) {
3.    // Step 1. rank suspicious combinations
4.    identify the set π′ of t-way suspicious combinations in F
5.    if ( π′ != empty  && ( |π′| < |π| ){
6.      π = π′
7.      produce a ranking R of all the t-way combinations in π
8.      // Step 2. generate new tests
9.      generate a set F' of new tests
10.     F = F ∪ F'
11.   }
12.   else if ( π′= empty )
13.     return an empty set of rankings;
14.   if (|π′| = |π|
15.       || any combination marked as inducing) {
16.     // Step 3. derive smaller combinations
17.     derive R1, R2, ... Rt based on π′
18.     return {R1, R2, ... Rt}
19.   }
20. } // end of while
```

Figure 1. Algorithm for identifying inducing combinations

The motivation for the third factor is that components of inducing combinations tend to appear more frequently in suspicious combinations. For example, assume that combination $c = (a \leftarrow 0, b \leftarrow 0)$ is inducing. Let $f = (a \leftarrow 0, b \leftarrow 0, c \leftarrow 0, d \leftarrow 0)$ be a test case. Test case f fails as it contains c . Let $f' = (a \leftarrow 1, b \leftarrow 1, c \leftarrow 0, d \leftarrow 0)$ be another test case, which passed since it does not contain c. The set of suspicious combinations derived from these two test cases is

$$\pi = \{(a \leftarrow 0, b \leftarrow 0), (a \leftarrow 0, c \leftarrow 0), (a \leftarrow 0, d \leftarrow 0), (b \leftarrow 0, c \leftarrow 0), (b \leftarrow 0, d \leftarrow 0)\}$$

In this set, the frequencies of $a \leftarrow 0$ and $b \leftarrow 0$ both are greater than others. The reason is that $(c \leftarrow 0, d \leftarrow 0)$ appears in f', which is a passed test case.

Suspiciousness of combination (ρ_c): Suspiciousness of a combination c is defined to be the average of suspiciousness of components that appear in c. Formally suspiciousness of combination c, $\rho_c(c)$ is computed by

$$\rho_c(c) = \frac{1}{[c]} \sum_{\forall o \in c} \rho(o) \qquad (2)$$

Suspiciousness of Environment (ρ_e): The environment of a combination c in a test f includes all components that appear in f but do not appear in c. The suspiciousness of the environment of a combination c in a test f is the average suspiciousness of the components in the environment of c. If

there is more than one (failed) test containing c in a test set, the suspiciousness of the environment of c in this test set is the minimum suspiciousness of environment of c in all the tests containing c. Formally, suspiciousness of the environment is computed by

$$\rho_e(c) = \text{Min}\left(\sum_{o \in f \wedge o \notin c} \rho(o),\ \forall f \in F\right) \quad (3)$$

Now we discuss how to actually rank the suspicious combinations based on ρ_c and ρ_e. Intuitively, the higher the value of ρ_c, the lower the value of ρ_e, the higher a combination is ranked.

To produce the final ranking, we first produce two rankings R_c and R_e of suspicious combinations, where R_c is in the non-descending order of ρ_c and R_e is in the non-ascending order of ρ_e. The final ranking R is produced by combining R_c and R_e as follows. Let c and c' be two suspicious combinations. Assume that c has ranks r_c and r_e in R_c and R_e, respectively, and c' has ranks r'_c and r'_e in R_c and R_e, respectively. In the final ranking R, c is ranked before c' if and only if $r_c + r_e < r'_c + r'_e$.

C. Test generation

The step of test generation is responsible for generating new test cases for a predefined number of top suspicious combinations. These new test cases are used to refine the ranking of suspicious combinations in the next iteration. Let c be a suspicious combination. A new test f is generated for c such that f contains c and the suspiciousness of the environment for c is minimized in f. When such a test case passes, this combination is removed from the suspicious set. When such a test case fails, the failure is more likely due to this combination since the suspiciousness of its environment is minimized.

One algorithm to find a new test case with minimum ρ_e for a suspicious combination is to generate all possible tests containing this combination, remove tests which already exist in F, and then select one with minimum ρ_e. This algorithm is very expensive. We next describe a more efficient, but heuristic, algorithm.

First, we generate a base test f as follows. For each parameter involved in c, we give the same value in f as in c. Doing so makes sure that f contains c. For each parameter in the environment of c, i.e., each parameter that is not involved in c, we choose a value (or component) whose suspiciousness ρ is the minimum. If there is more than one value with minimum ρ, one of them is selected randomly.

Next, we check whether the base test f is really new, i.e., making sure that f has not been executed before. If so, f is returned as the new test that contains c and has minimum ρ_e. If not, we pick one parameter randomly and change its value to a value with the next minimum ρ. Again, this test is checked to see whether it is a new test. These steps are repeated until a new test is found, or the number of attempts for finding new test case reaches a predefined number. In the latter case, the combination c is marked as an inducing combination, because it is very likely that all the test cases containing this combination have been executed (and all of them must have failed).

D. Reduction

In the step of reduction, the set of t-way suspicious combinations is analyzed to derive suspicious combinations of smaller size, i.e., size 1 to $t-1$. A k-way combination c, where $1 \le k \le t-1$, is suspicious if all the (k+1)-way combinations containing c are suspicious.

Our reduction algorithm works as follows. A bucket is assigned to each $(t-1)$-way combination c to hold t-way suspicious combinations that contain c. For each t-way suspicious combination in π, we put it into t buckets, one for each $(t-1)$-way combination that it contains. A $(t-1)$-way combination c is identified to be a suspicious combination if the number of t-way combinations in its bucket is equal to the number of all possible t-way combinations containing c.

After all the $(t-1)$-way suspicious combinations are identified, they are ranked using the same algorithm for ranking t-way suspicious combinations.

The similar process can be applied to derive suspicious combinations of size $t-2$, and so on, until we derive suspicious combinations of size 1.

E. Stopping condition

There are three stopping conditions in Fig. 1. The first condition is that π becomes empty. This situation occurs when all inducing combinations are of size greater than t. In this situation, assumption 3 is not satisfied. In this situation, no rankings of suspicious combinations are produced.

The second condition is that the size of π does not change from the previous iteration. This situation occurs when all the new tests generated in the previous iteration fail, and thus no suspicious combination is removed from π.

The third stopping condition is that the framework finds a suspicious combination marked as an inducing combination. These combinations are marked in test generation step, when no new test is found for them.

F. Discussion

Our approach is by nature heuristic. On one hand, suspicious combinations that are ranked top by our approach may not be truly inducing. On the other hand, truly inducing combinations may not be ranked top by our approach.

While our approach focuses on analyzing t-way combinations, it guarantees to identify inducing combinations involving no more than t parameters to be suspicious combinations. Let c be an inducing combination, we consider the following two cases.

Case (1): c is a t-way combination. As the initial test set is a t-way test set, there is at least one test that contains c, and all test cases containing c must fail, since c is inducing. Therefore, c is identified to be a suspicious combination by our approach.

Case (2): The size of c is less than t. All t-way combinations containing c are inducing combinations and are identified to be suspicious combinations. Hence, the reduction step identifies c as a suspicious combination.

Note that when an inducing combination involves more than t parameters, it may not appear in the initial t-way test set, and our algorithm does not identify it to be a suspicious combination.

G. Complexity analysis

Let k be the number of parameters, d the largest domain size and n the number of test cases in the test set. The maximum number of t-way combinations is $m = \binom{k}{t}d^t$.

The rank generation step needs to sort the set of suspicious combinations for three times, once for each ranking R_c, R_e, and R. The sorting dominates the complexity of this step, which is $O(m * \log m)$.

The test generation step needs to select $(k - t)$ values with minimum ρ first, which takes $(k - t) * O(d)$. Then it needs to check whether it is new, which is $O(k * n)$. In the worst case, a new test is not found after a predefined number of attempts. Thus the complexity for this step is $(k - t) * O(d) * O(k * n)$.

In the reduction step, each t-way suspicious combination is put into t buckets. It takes $O(t)$ to determine whether a t-way combination belongs to a particular bucket. There are $l = \binom{k}{t-1}d^{t-1}$ buckets. So the complexity for all t-way combinations is $O(t * l * m)$. This computation is performed for 1 to $(t - 1)$-way combinations, and the total complexity is $O(t^2 * l * m)$. The reduction step ranks suspicious combinations, which is however dominated by finding suspicious combinations.

IV. Example

In this section, we illustrate our approach using an example program, which is shown in Fig. 2. Method *foo* has a fault in line 6. The correct statement should be $r+= (b - d)/(a + 2)$, but operator "+" is missing. The input parameter model consists of $P = \{a, b, c, d\}$, and $d_a = \{0,1\}$, $d_b = \{0,1\}$, $d_c = \{0,1,2\}$, and $d_d = \{0,1,2,3\}$. The faulty statement is reachable with a test f such that (1) f(a) = 0; and (2) f(c) = 0 or f(d) = 3. So the inducing combinations are $(a \leftarrow 0, c \leftarrow 0)$ and $(a \leftarrow 0, d \leftarrow 3)$.

Suppose that the program is tested by a two-way test set. The result of the test executions is shown in Table I, where 3 out of 12 tests fail. Test cases #1 and #7 fail because they contain combination $(a \leftarrow 0, c \leftarrow 0)$. Test case #10 fails because it contains $(a \leftarrow 0, c \leftarrow 0)$ and $(a \leftarrow 0, d \leftarrow 3)$.

Our approach takes Table I as input. Nine suspicious two-way combinations are identified, and are listed in the first column of Table II. Then our approach computes the suspiciousness of all the components (seven) that appear in a suspicious combination.

For example, component $c \leftarrow 0$ appears in 3 failed test cases while there are 3 failed test cases, so $u(c \leftarrow 0) = 1$. The frequency of $c \leftarrow 0$ in the test set is 4, so $v(c \leftarrow 0) = 3/4$; 5 out of 9 members of suspicious combinations set

```
1  public static int foo(int a,int b, int c,int d){
2      int r = 1;
3      b += a + c;
4      switch (a){
5              case 0 :
6                  if (c<1 || d>2)
7                      //r += (b-d)/(a+2);
8                      //fault : + is missing;
9                      r = (b-d)/(a+2);
10                 else
11                     r = b/(c+2);
12                 break;
13             case 1 :
14                 r = c*(a-d);
15                 break;
16         }
17     return r;
18 }
```

Figure 2. An example of faulty program

TABLE I. TWO-WAY TEST SET AND STATUS

Test #	a	b	c	d	Status
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

TABLE II. SUSPICIOUS COMBINATIONS AND THEIR CORRESPONDING VALUES

Suspicious Combination	ρ_c	R_c	ρ_e	R_e	$R_c + R_e$	R	New test case	Status
$a \leftarrow 0, c \leftarrow 0$	0.6713	1	0.2460	1	2	1	$(a \leftarrow 0, b \leftarrow 0, c \leftarrow 0, d \leftarrow 1)$	fail
$b \leftarrow 1, c \leftarrow 0$	0.6176	2	0.4352	3	5	2	$(a \leftarrow 1, b \leftarrow 1, c \leftarrow 0, d \leftarrow 1)$	pass
$c \leftarrow 0, d \leftarrow 0$	0.5324	4	0.3849	2	6	3	$(a \leftarrow 1, b \leftarrow 0, c \leftarrow 0, d \leftarrow 0)$	pass
$c \leftarrow 0, d \leftarrow 3$	0.5509	3	0.5204	4	7	4	$(a \leftarrow 1, b \leftarrow 0, c \leftarrow 0, d \leftarrow 3)$	pass
$c \leftarrow 0, d \leftarrow 2$	0.5324	4	0.5204	4	8	5	$(a \leftarrow 1, b \leftarrow 0, c \leftarrow 0, d \leftarrow 2)$	pass
$a \leftarrow 0, d \leftarrow 3$	0.4537	5	0.6176	5	10	6	$(a \leftarrow 0, b \leftarrow 0, c \leftarrow 2, d \leftarrow 3)$	fail
$b \leftarrow 1, d \leftarrow 3$	0.4000	6	0.6713	6	12	7	$(a \leftarrow 1, b \leftarrow 1, c \leftarrow 1, d \leftarrow 3)$	pass
$b \leftarrow 1, d \leftarrow 2$	0.3815	7	0.6713	6	13	8	$(a \leftarrow 1, b \leftarrow 1, c \leftarrow 1, d \leftarrow 2)$	pass
$b \leftarrow 0, d \leftarrow 0$	0.2460	8	0.6713	6	14	9	$(a \leftarrow 1, b \leftarrow 0, c \leftarrow 2, d \leftarrow 0)$	pass

contain $c \leftarrow 0$, so $w(c \leftarrow 0) = 5/9$. The computations for all components are as follows:

$$\rho(c \leftarrow 0) = \frac{1}{3} * \left(1 + \frac{3}{4} + \frac{5}{9}\right) = 0.7685$$

$$\rho(d \leftarrow 0) = \frac{1}{3} * \left(\frac{1}{3} + \frac{1}{3} + \frac{2}{9}\right) = 0.2963$$

$$\rho(d \leftarrow 2) = \frac{1}{3} * \left(\frac{1}{3} + \frac{1}{3} + \frac{2}{9}\right) = 0.2963$$

$$\rho(d \leftarrow 3) = \frac{1}{3} * \left(\frac{1}{3} + \frac{1}{3} + \frac{3}{9}\right) = 0.3333$$

$$\rho(b \leftarrow 0) = \frac{1}{3} * \left(\frac{1}{3} + \frac{1}{7} + \frac{1}{9}\right) = 0.1958$$

$$\rho(b \leftarrow 1) = \frac{1}{3} * \left(\frac{2}{3} + \frac{2}{5} + \frac{3}{9}\right) = 0.4667$$

$$\rho(a \leftarrow 0) = \frac{1}{3} * \left(1 + \frac{3}{6} + \frac{2}{9}\right) = 0.5741$$

According to formula (2), ρ_c for a suspicious combination c is the average suspiciousness of the components that c contains. For example, in combination $(a \leftarrow 0, c \leftarrow 0)$, ρ_c is $(0.5741 + 0.7685)/2 = 0.6713$. After computing ρ_c for all suspicious combinations, we ranked them based on the non-ascending order of ρ_c. The values of ρ_c and R_c for each suspicious combination are shown in the second and third columns of Table II.

Next we compute ρ_e for each suspicious combination using formula (3). For example, there are three test cases, test #1, test #7, and test #10, that contain $(a \leftarrow 0, c \leftarrow 0)$. Therefore,

$$\rho_e(a \leftarrow 0, c \leftarrow 0) = \min\left(\left(\frac{\rho(b \leftarrow 0) + \rho(d \leftarrow 0)}{2}\right) = 0.2460, \left(\frac{\rho(b \leftarrow 1) + \rho(d \leftarrow 2)}{2}\right) = 0.3815, \left(\frac{\rho(b \leftarrow 1) + \rho(d \leftarrow 3)}{2}\right) = 0.4000\right) = 0.2460$$

Next we rank suspicious combinations by a non-descending order of ρ_e, as shown in column R_e of Table II.

Finally, the two rankings in columns R_c and R_e are combined to produce a final ranking of the suspicious components (column R). In this final ranking, inducing combination $(a \leftarrow 0, c \leftarrow 0)$ is ranked on the top, and the other $(a \leftarrow 0, d \leftarrow 3)$ is ranked 6th.

Then new tests are generated for the most suspicious combinations. For suspicious combination $(a \leftarrow 0, c \leftarrow 0)$, we assign values to parameters in its environment, i.e., b and d, such that the suspiciousness of each value is minimum. For b, 0 is selected, as $\min(\rho(b \leftarrow 0) = 0.1958, \rho(b \leftarrow 1) = 0.4667) = 0.1958$. For d, 1 is selected as $\min(\rho(d \leftarrow 0) = 0.2963, \rho(d \leftarrow 1) = 0, \rho(d \leftarrow 2) = 0.2963, \rho(d \leftarrow 3) = 0.3333) = 0$. So a new test $(a \leftarrow 0, b \leftarrow 0, c \leftarrow 0, d \leftarrow 1)$ is generated.

In this example, we generate a new test case for each suspicious combination since there are only nine combinations. As shown in the last column of Table II, all the new test cases pass except two that contain two inducing combinations. In the next iteration, the combinations that appear in a passed test case are not suspicious anymore. Therefore, all combinations except two inducing combinations are removed from the suspicious combinations set, and this set consists of the two combinations, which are both inducing combinations.

Note that this example represents a best case scenario of our approach. In the next section, we provide an experimental evaluation of our approach.

V. EXPERIMENT

We built a prototype tool called BEN that implements our approach. (BEN is a Chinese word that means "root cause".) We used this tool to conduct an experiment on a set of six benchmark programs.

A. Experimental design

1) Subject programs

We used six C programs, *count*, *series*, *tokens*, *ntree*, *nametbl*, and *cmdline*, as subject programs [7]. Each of these programs contains some faults. To determine whether a test case fails or passes, we created a fault-free version of each program according to the accompanying fault descriptions.

In combinatorial testing, the result may be different by different ways of modeling the input space. To reduce bias, we used the same models for the six programs as in previous work [13].

Table III shows properties of subject programs and their input models. The second column (*LOC*) shows the number of lines of uncommented code in these programs. The third column shows the number of faults. The last column (*Input Model*) shows the input parameter model of each program, which includes the number of parameters and their domain size. We represent it by $(d_1^{k_1} \times d_2^{k_2} \times \ldots)$, where $d_i^{k_i}$ indicates that there are k_i number of parameters with domain size as d_i. Note that $k_1 + k_2 + \cdots = k$, which is the total number of parameters. For example, *count* has six parameters, among which two parameters have a domain size of two, and four parameters have a domain size of three. More details about these models can be found elsewhere [13].

Each subject program contains multiple faults. Generally speaking, the more faults, the more failure-inducing combinations, and the easier it is to find them. To make the problem more challenging, two additional versions for each program are created; a version with about 50% of faults, and a version with a single fault. We refer to these versions by versions1,2, and 3 respectively. Then we run the tool three times for each program, once for each version.

TABLE III. SUBJECT PROGRAMS

	LOC	# of faults	Input model
count	42	8	$(2^2 \times 3^4)$
series	288	4	$(2^1 \times 4^2 \times 6^1)$
tokens	192	5	$(2^2 \times 3^2)$
ntree	307	8	(4^4)
nametbl	329	8	$(2^1 \times 3^2 \times 5^2)$
cmdline	336	9	$(2^1 \times 3^4 \times 4^1 \times 6^2 \times 15^1)$

2) Metrics

To measure the effectiveness of our approach, we compute the percentage of truly inducing combinations in the top 10 ranked suspicious combinations. If a combination in top 10 is not inducing, we also compute its inducing probability.

We measure the efficiency of BEN by the percentage of new test cases generated and number of iterations needed.

For the purpose of the evaluation, in order to detect truly inducing combinations, we run the exhaustive test set. A combination is truly inducing if all possible tests containing this combination fail.

3) Test Generation

The initial t-way test set is generated using the ACTS tool [1]. When we generate new tests, we generate a new test for each of the top 10 suspicious combinations.

B. Results and discussion

We conduct the experiment by taking a 2-way test set as the initial test set, except for version 3 of *series*, where both the 2-way and 3-way tests are used. The reason is that there is no 2-way inducing combination for version 3 of *series*.

The results of our experiment are summarized in Table IV. We will not explain the column headers one by one, as they are self-explanatory. We point out that the 7th column (*ratio of inducing combinations to all combinations*) is intended to show the difficulty of the identification problem. Typically, the fewer inducing combinations, the more effort needed to identify them.

For example, in version 3 of *cmdline*, there are only 2 inducing combinations out of 836 possible 2-way combinations. In version 1 of *count*, every combination is inducing. It is easy to see that identifying inducing combinations in version 1 of count is much easier than in version 3 of *cmdline*.

Note that the results for versions 1 and 2 of *tokens* are the same. Version 2 was created by removing 3 out of 5 faults of version 1. However, both versions produce the same output. The reason is that we used the same model as previous work [13], which does not capture the difference between these two versions. Columns 8, 9, and 10 (*#of iterations*, *#of new test cases,* and *percentage of executed tests to the exhaustive test set*) are intended to show the efficiency of our approach. In general, a small percentage of tests need to be executed by our approach. There are a few cases where more than 50% tests were executed. One case is for version 3 of *series* with 3-way test set, we ran 60 % of the test cases, i.e., 116 test cases. However, only 10 new test cases were added by our approach and the other 106 tests were in the initial test set. The other cases are for the three versions of *tokens*. This program has a small number of the exhaustive test cases(36 test cases), and there were 9 tests in the initial test set.

In contrast, for the three different versions of *cmdline*, the largest program, at most 0.044% of possible test cases were executed, but as shown in Fig. 3, later, we can

TABLE IV. EXPERIMENTAL RESULTS FOR T-WAY COMBINATIONS

Program	Version	Size of exhaustive test set	Size of Initial Test Set	# of all t-way combinations	# of inducing combinations	Ratio of inducing combinations to all combinations	# of iterations	# of new test cases	percentage of executed tests to exhaustive test set	# of suspicious combinations	Percentage of inducing combinations in top 10 suspicious combinations
count	1	324	12	106	106	1	2	10	7%	106	100%
	2				30	0.2830	4	30	13%	45	100%
	3				13	0.1226	2	10	7%	20	100%
series	1	192	24	92	10	0.1087	3	20	23%	32	50%
	2				2	0.0217	4	22	24%	4	50%
	3				0	0	4	6	16%	0	-
			106*	224	6	0.0268	2	10	60%	12	60%
tokens	1	36	9	37	14	0.3784	2	10	53%	21	100%
	2				14	0.3784	2	10	53%	21	100%
	3				8	0.2162	2	10	53%	14	80%
ntree	1	256	16	96	24	0.2500	2	10	10%	48	100%
	2				14	0.1458	3	20	14%	44	60%
	3				2	0.0208	4	22	15%	2	100%
nametbl	1	450	25	126	83	0.6587	2	10	8%	105	100%
	2				30	0.2381	2	10	8%	74	100%
	3				6	0.0476	10	83	24%	6	100%
cmdline	1	349920	95	836	252	0.3014	4	30	0.036%	568	50%
	2				197	0.2356	7	60	0.044%	463	70%
	3				2	0.0024	4	24	0.034%	4	50%

* three-way

TABLE V. INDUCING PROBABILITIES OF TOP 10 SUSPICIOUS COMBINATIONS THAT ARE NOT INDUCING

Program	Version	Rank	# of possible test cases containing combination	# of possible failed test cases containing this combination	Inducing probability
series	2	1	12	10	0.8333
		2	12	10	0.8333
ntree	2	1	16	15	0.9375
		3	16	15	0.9375
		7	16	13	0.8125
		10	16	15	0.9375
cmdline	1	1	29160	27216	0.9333
		7	11664	10674	0.9151
		8	43740	40824	0.9333
		9	11664	10674	0.9151
		10	29160	25704	0.8815
	2	4	11664	11661	0.9997
		7	11664	11661	0.9997
		9	11664	11655	0.9992
	3	3	7776	5832	0.75
		4	3888	2766	0.7114

still rank all the inducing combinations to the top 10.

The last two columns (*# of suspicious combinations* and *percentage of inducing combinations in top 10 suspicious combinations*) show the effectiveness of our approach. For 10 (out of 18) versions of these programs, all the top 10 ranked suspicious combinations are truly inducing. For other versions, Table V shows the ranks and inducing probabilities of the top 10 ranked suspicious combinations that are not truly inducing. All of these combinations have a very high inducing probability. The lowest inducing probabilities happen in version 3 of *cmdline*, where the 3rd and 4th ranked combinations have an inducing probability of 0.75 and 0.7114, respectively. The highest inducing probabilities happen in version 2 of *cmdline*, where the 4th, 7th, and 9th ranked combinations have an inducing probability that is close to 1, even if they are not truly inducing.

In two versions of *series,* 1st and 3rd (with 3-way test set), and the 3rd version of *tokens,* the third stopping condition is satisfied, and truly inducing combinations are found. So the information of other suspicious combinations is excluded from Table V.

As it is shown in Table IV, the set of suspicious combinations becomes empty in version 3 of *series* where 2-way test set is applied. In other cases, reaching the stable point, satisfying the second condition, happens.

The reduction step finds 1-way suspicious combinations in 13 (out of 18) versions; in 8 versions of theses 13 versions, all of top ranked combinations are inducing (Table VI). For other 5 versions, the ranks and inducing probabilities of non-inducing but suspicious combinations are shown in Table VII.

TABLE VI. EXPERIMENTAL RESULTS FOR (T-1)-WAY COMBINATIONS

Program	Version	# of all (t – 1)-way combinations	# of inducing combinations	Ratio of inducing combinations to all combinations	# derived combinations	Percentage of inducing in top 10 derived combinations
count	1	16	16	1	16	100%
	2		2	0.125	2	100%
	3		1	0.0625	1	100%
series	1	16	0	0	1	0%
tokens	1	10	2	0.2	2	100%
	2		2	0.2	2	100%
	3		1	0.1	1	100%
ntree	1	16	2	0.125	2	100%
	2		0	0	1	0%
nametbl	1	18	7	0.3889	10	70%
	2		2	0.1111	2	100%
cmdline	1	45	7	0.1556	13	50%
	2		6	0.1333	7	85%

The charts in Fig. 3 show the distribution of inducing and non-inducing combinations in the ranking of suspicious combinations after each iteration. Due to limited space, we only show the distribution for the 3rd version of each program, except for program *series*, where version 2 is shown. The vertical axis shows the number of iterations. The horizontal axis shows the ranks. Inducing and non-inducing combinations are shown by different colors.

The charts show that our approach can quickly rank all the inducing combinations to the top. For example, *nametbls* has 41 suspicious combinations in the first iteration. There are 6 truly inducing combinations, which are ranked 1 to 4, 7, and 11. In the second iteration, we have 33 suspicious combinations, and 5 out of 6 inducing combinations are ranked to the top 5. In the third iteration, all 6 inducing combinations are ranked to the top 6. Although BEN runs 10 iterations to reach to the stable point, all 6 inducing combinations come to the higher ranks sooner than 10 iterations.

Note that version 2 of *series* only has two inducing combinations, and they are ranked in the 3rd and 4th place.

C. Threats to validity

Threats to internal validity are other factors that may be responsible for the experimental results, without our knowledge. We have tried to automate the experimental procedure as much as possible, as an effort to remove human errors. In particular, we build clean versions for all six subject programs, and a tool that automatically compares the results of the clean version and a faulty version to determine truly inducing combinations. Further, consistency of the results has been carefully checked to detect potential mistakes made in the experiment.

TABLE VII. INDUCING PROBABILITIES OF TOP 10 (T-1)-WAY SUSPICIOUS COMBINATIONS THAT ARE NOT INDUCING

Program	Version	Rank	# of possible test cases containing combination	# of possible failed test cases containing this combination	Inducing probability
series	1	1	32	30	0.9375
ntree	2	1	64	61	0.9531
nametbl	1	5	90	86	0.9556
		8	90	80	0.8889
		9	90	86	0.9555
cmdline	1	2	23328	22338	0.9576
		3	23328	22338	0.9576
		6	23328	22338	0.9576
		7	23328	22338	0.9576
		10	23328	22368	0.9588
	2	5	23328	23302	0.9988

Threats to external validity occur when the experimental results could not be generalized to other programs. We use subject programs from previous work [7]; these programs are created by a third party and have been used in other studies [16]. But the subject programs are programs of relatively small size with seeded faults. More experiments on larger programs with real faults can reduce external validity of our findings.

The original versions of the subject programs had multiple faults, and thus many inducing combinations. So they could be identified more easily than those for programs with a small number of inducing combinations. To mitigate this threat, we conduct our experiment on 3 versions of each program, with all faults, 50% of faults, and one fault, respectively.

VI. RELATED WORK

Delta debugging [12] is a technique that tries to find a minimum set of failure-inducing input values in a failed test. It involves systematically changing or removing the values in a failed test to create new tests. Two similar techniques, called FIC and FIC_BS [13], try to identify all the faulty interactions contained in a failed test. The notion of a faulty interaction is the same as the notion of an inducing combination defined in this paper. FIC and FIC_BS assume that no new inducing combinations are introduced when a value is changed to create a new test.

Our approach is different from these previous techniques in that we try to identify inducing combinations in a combinatorial test set, instead of a single failed test. On one hand, a test set contains more information than a single test. On the other hand, doing so makes it possible to identify inducing combinations that appear in different tests. Moreover, the assumption made by FIC and FIC_BS may not hold for many applications, as changing a value in a test introduces many new combinations, and assuming that all of them are non-inducing is over-optimistic.

Yilmaz et al. [11] proposed a machine learning approach to identify likely inducing combinations from a given combinatorial test set. Their approach builds a data structure called classification tree, and assigns a score to each likely inducing combination. A combination is classified to be an inducing combination if its score is greater than a threshold value. This approach is used to guide the generation of new tests in an adaptive combinatorial testing technique [4].

The preceding approach identifies inducing combinations based on a combinatorial test set only, i.e., without adding new tests. Considering that a combinatorial test set is often produced such that it contains as few tests as possible while still achieving t-way coverage, many

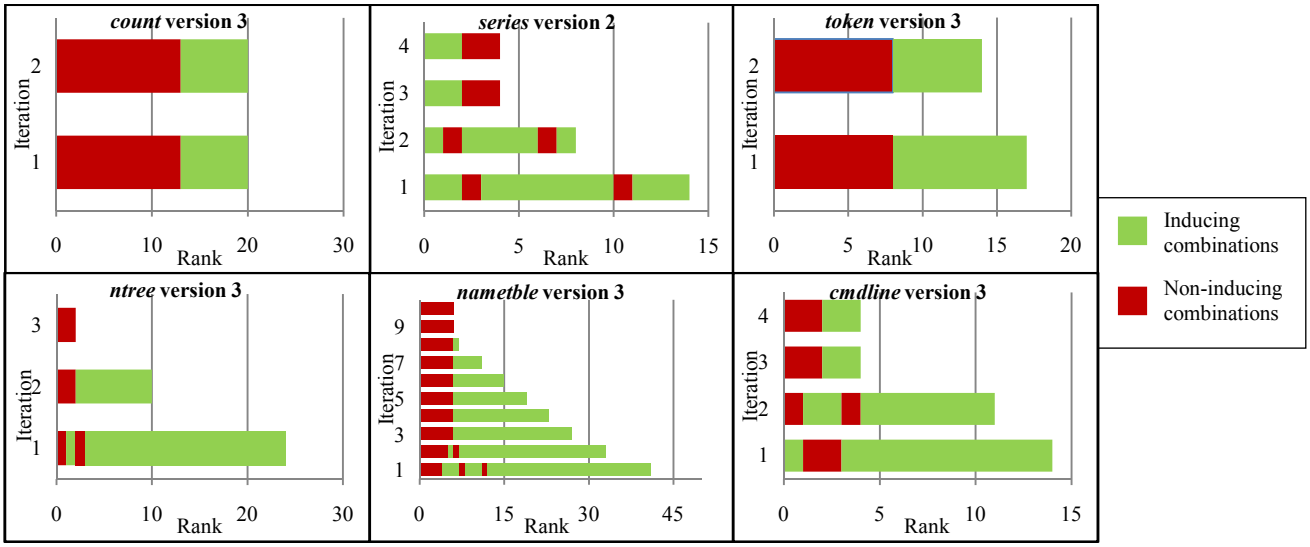

Figure 3. Distribution of inducing and non-inducing combinations in suspicious set

combinations are covered only once in a combinatorial test set. As a result, a combinatorial test set alone often provides insufficient information for effective classification, especially when there are a large number of inducing combinations and failed test cases.

The work that is mostly related to ours is a technique called AIFL [8, 9]. Given a combinatorial test set, AIFL first identifies all the suspicious combinations, i.e., combinations that only appear in failed tests. Then, for each failed test, AIFL generates k test cases by changing the value of one parameter at a time, where k is the number of parameters. The new value of a changed parameter could be any value in its domain. New tests are then used to refine the set of suspicious combinations. At this point, AIFL stops and outputs the set of suspicious combinations. InterAIFL [10] extends AIFL by adopting an iterative framework. That is, new tests are generated to refine the set of suspicious combinations iteratively until a fixed point is reached, i.e., the set of suspicious combinations becomes stable.

Our approach identifies suspicious combinations in the same way as AIFL and Inter-AIFL. However, our approach goes one step further to produce a ranking of suspicious combinations. This ranking helps the debugging effort to focus on the most suspicious combinations. Our approach also differs significantly in the way of generating new tests: our test generation is based on the notions of suspiciousness combination and suspiciousness of environment.

VII. CONCLUSION

In this paper, we presented an approach to identifying failure-inducing combinations in a combinatorial test set. Our approach adopts an iterative framework that ranks suspicious combinations and generates new tests repeatedly until a stable condition is reached. The novelty of our approach lies in the fact that we rank suspicious combinations and generate new tests based on the notions of suspiciousness of a combination and suspiciousness of its environment. The higher the suspiciousness of a combination, the lower the suspiciousness of its environment, the higher this combination is ranked. New tests are generated for a user-defined number of most suspicious combinations such that the suspiciousness of the environment of a combination is minimized in each test. Our experimental results show that our approach is very effective in terms of quickly identifying and ranking failure-inducing combinations to the top.

There are two major directions to continue our work. First, we plan to conduct more empirical studies to further evaluate the performance of our approach. In particular, we plan to apply our approach to larger and more complex programs. Second, this work is part of a larger effort to develop fault localization techniques that leverage the result of combinatorial testing. The next step in our project is to go inside the source code and find a particular line or block of code that contains the fault. We believe that failure-inducing combinations provide important insights about how different parameters interact with each other and can be used to reduce the scope of the code that needs to be analyzed in the next step.

VIII. ACKNOWLEDGMENT

We thank Christopher Lott for providing the benchmark programs and Jian Zhang for allowing us to use their combinatorial models. This work is supported by two grants (70NANB9H9178 and 70NANB10H168) from Information Technology Lab of National Institute of Standards and Technology (NIST) and a grant (61070013) of National Natural Science Foundation of China.

Disclaimer: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

IX. REFERENCES

[1] Advanced Combinatorial Testing System (ACTS), 2010. http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html.

[2] 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.

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

[4] E. Dumlu, C. Yılmaz, M. B. Cohen, and A. Porter. Feedback driven adaptive combinatorial testing. In Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2011), pages 243-253, 2011.

[5] D.R. Kuhn, D.R. Wallace, A.M. Gallo. Software fault interactions and implications for software testing. IEEE Transaction on Software Engineering, 2004, 30: 418–421

[6] Y. Lei, R. Kacker, D. Kuhn, V. Okun, J. Lawrence, IPOG/IPOD: Efficient test generation for multi-way software testing, Journal of Software Testing, Verification, and Reliability, 18(3):125-148, Sept. 2008.

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

[8] C. Nie, H. Leung, and B. Xu. The minimal failure-causing schema of combinatorial testing. ACM Transactions on Software Engineering and Methodology, Volume 20 Issue 4, September 2011.

[9] L. Shi, C. Nie, B. Xu. A software debugging method based on pairwise testing. In Proceedings of the International Conference on Computational Science (ICCS2005), pages 1088-1091, 2005.

[10] 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, 2010.

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

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

[13] Z. Zhang, and J. Zhang. Characterizing failure-causing parameter interactions by adaptive testing. In Proceding of ACM International Symposium on Software Testing and Analysis (ISSTA 2011), pages 331-341, 2011.