Identification of Failure Inducing Combinations using Constraint Satisfaction Algorithm

Mahipal Chakravarthy G, Keerthana V, Abhav Garg, Preeti Satish, D R Ramesh Babu, Krishnan Rangarajan
Department of Computer Science and Engineering
Dayananda Sagar College of Engineering
Bangalore ,India
mahipalchakravarthy@gmail.com, keerthanareddyv1234@gmail.com, abhavgarg@hotmail.com, preetisatish8@gmail.com, bobrammysore@gmail.com, krishnanr1234@gmail.com

Abstract—**The successful eminence of combinatorial testing can be realized with effective fault localization techniques.In this paper, we present an effective fault localization technique that can accurately pin down the FICs (Failure Inducing Combinations) using constraint satisfaction approach. The importance of the technique comes from the fact that given an executed test suite as input, it localizes the FICs by catering to all-t-way combinations while generating only one additional test case for each FIC. Our initial study and experimentation are found to be promising providing scope for rigorous empirical study.**

Keywords—Fault inducing combinations,Combinatorial testing,

I. Introduction

Combinatorial testing is used to detect software failures caused by interaction among various input parameters. For instance, an interaction leading to failure is shown in the code snippet.

```
if (temperature < 20)
{
   //normal code
         if (volume > 500)
                  { //code that triggers fault }
         else{//correct code }
}
else {//normal code }
```

Note that the failure is triggered in the above snippet only when both the parameters temperature < 20 and volume > 500 are true.

When such a failure is detected, we need to find the combination of parameters that have caused said failure and fix it. This is called fault localization. There are multiple tools that are already in place that help us with fault localization like BEN and MIXTGTE. Our approach differs from these methods in the following ways.

- BEN generates only specific t-way combinations but our approach generates all possible t-way combinations.
- MIXTGTE generate all the possible additional test cases in pinning down a FICs,but our approach generates only one additional test case for each FIC.

True FIC's are those that always generate failures when the said parameters are being introduced. In our paper, we work towards the identification of true FIC's as well as generating the probability a non-true FIC producing failures.

For this to work, we first have an input set of executed test cases which consist of parameters and pass/fail value. Once the input set is read, the set of passed test cases and failed test cases are distinguished from each other. All the combinations in the test cases that have passed are tagged as non-FIC's whereas the combinations in the failed test cases are to be worked upon. Likely FIC's are found for the failed test cases and additional test cases are generated taking the likely FIC's as a subset for each test case. Generate additional test cases which have the least probability of failure compared to all the possible test cases with the current combination. Once the test cases are generated, the pass/fail value of the test cases are observed and if there is any pass value, the combination is not a FIC. Otherwise, the combination is termed as a failure-inducing combination and a new test is run with a different subset until no further tests can be run. At the end of this iterative process, if there still remains a combination that is generating failures, that combination is a true-FIC. This means that each time that particular combination of parameters are used, the result of the test case will 100% be fail.

II. Related Work

Two techniques, called FIC and FIC_BS [1], try identifying all the inducing combinations present in a failing test. These approaches take one failing test from a combinatorial test set, then generate and execute a small number of tests in a systematic manner to identify inducing combinations in the failing test. New tests are generated such that one value of the failing test is changed to another possible value. When the newly generated test passes, the initial value is part of an inducing combination because its removal makes the test pass. FIC generates k tests, where k is the number of parameters, for each FIC.

FIC_BS is the binary search version of FIC. To generate a new test, FIC_BS changes the values of k/2 parameters of the failing test. If the newly generated test passes, FIC_BS searches for inducing combinations in the changed values (k/2). The process continues until all inducing combinations are found. FIC and FIC_BS assume that no new inducing combinations are introduced when a value is changed to create a new test.

Li et al. [2] introduced two approaches, RI and SRI, for identifying inducing combinations. These techniques use a method called delta debugging [3] in an iterative framework. The RI approach takes one failing test from the initial combinatorial test set, and adopts a similar approach to FIC_BS to generate a small number of tests. The SRI approach is an improved version of RI, and it takes as input one failing test, f. Then it tries to generate a passing test similar to f. SRI uses the fact that the inducing combination appeared in the failing test f, but not in the similar passing test. Therefore, it focuses on the parameters, which are different in the failed and passing tests. SRI could identify inducing combination by generating fewer tests than RI.

The AIFL and InterAIFL approaches in [4][5] first identify a set A of suspicious combinations as candidates for being inducing. Second, it generates a group of tests for each failing test using the SOFOT strategy [6]. Let k be the number of parameters. For each test f, the SOFOT strategy generates k tests by changing the value of one parameter at a time. Each test is different from the original test f in one value; the value is selected randomly from the corresponding parameters domain. After executing the newly generated tests, combinations that appeared in the passing tests are removed from the suspicious set A.

Yilmaz et al. proposed a ML approach for finding failure-inducing combinations [7] which analyses the combinatorial test set and tests statuses while also building a classification tree which is used to predict inducing combinations. Shakya et al. in [8] made some improvements in identifying failure-inducing combinations based on this work.

Ghandehari et al. came up with an approach called BEN [9] which produces a ranking of statements in terms of their likelihood of being faulty by leveraging the result of combinatorial testing.

III. Approach

In this section we present the approach in detail as shown in the fig. 1.We explicate on each step of our approach with help of a synthetic example test suite mentioned in table 1.

TABLE-I EXAMPLE EXECUTED TEST SUITE

Test#	a	b	c	D	Status
1	1	2	1	0	Pass
2	1	1	0	1	Pass
3	2	3	1	0	Pass
4	1	3	1	0	Fail

A. Step 1- Inputs:

The algorithm takes the executed test suite as the input. This test suite is a collection of test cases with their execution result (pass/fail) where each test case contains the assigned values for the input parameters.

The basis for our algorithm is that a failure is caused by a specific combination of values in the inputs.For example, assume that the system under test has a set P with k input parameters, denoted by P = {p1, p2 , … , pk }. Let d_i be the domain of each parameter pi. That is, d_i contains all possible values that pi could take.

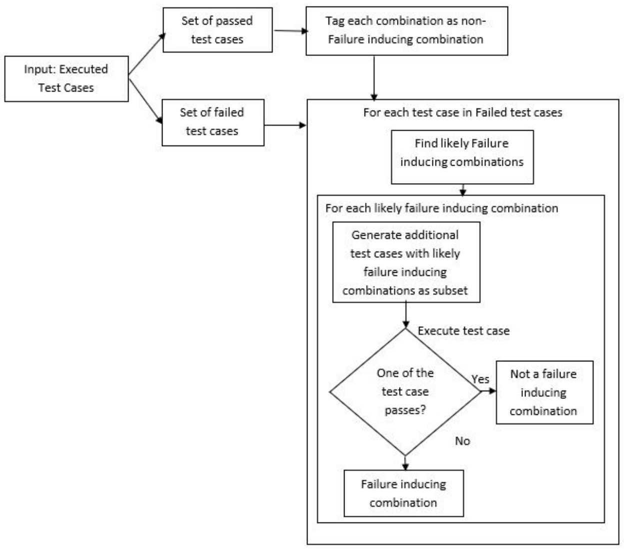

Input: Executed Test Cases
Set of passed test cases
Set of failed test cases
Tag each combination as non-Failure inducing combination
For each test case in Failed test cases
Find likely Failure inducing combinations
For each likely failure inducing combination
Generate additional test cases with likely failure inducing combinations as subset
Execute test case
One of the test case passes?
Yes
No
Not a failure inducing combination
Failure inducing combination

Fig. 1. Approach to pin down FICs

Let all the t-way parameter combinations in P be P_c={(p1),(p2),….,(p1,p2),…..,(pi,pk),……,(p1,p2, ….,pk)} i.e. for a set of k parameters, there will be (2^k-1) combinations.

Consider the Synthetic test case from table 1 as an example. From the table 1, the set of parameters is P={a,b,c,d} and d_a={1,2} d_b={1,2,3} d_c={0,1} d_d={0,1}

B. Step 2- Segregation and tagging of pass and fail test cases:

In this step we identify combinations in test cases as FIC, n-FIC, LFIC. This segregation is based on the constraint satisfaction problem. It is done as follows.

The input test set is then divided into two categories, a set of passed test cases and a set of failed test cases.

Let T= {v1, v2, …, vk} be a test case containing k input values. Accordingly, V_c contains all the combinations of values in T.

Let 'S' be a list containing {(P_{ci} ,V_{ci}) ->M) where P_{ci} is the parameter combination, V_{ci} is the value combination for the corresponding parameter in P_{ci} and M is a tag which can take one of three values 'Failure Inducing' , 'Non Failure Inducing' or 'Likely Failure Inducing'.

In the set of passed test cases, there will be no FICs. Therefore all the t-way combinations in this set are tagged as non-FIC (n-FIC). That is, for each element in V_c, add each combination (P_{ci},V_{ci}) as n-FIC to list S.

In the set of failed test cases, all the t-way combinations of each test case that are not in list S are tagged as Likely FIC(LFIC) and added to the suspicious list of that test case.

As a result, the suspicious list will now comprise of the following kinds of combinations:

1. The combination is failure inducing.
2. The combination consists of a FIC as its subset.
3. The combination is not failure inducing but was not tagged because there was no passed test case with the current combination

For the running example

The result of test case 1 is 'pass' hence all the t-way combinations {(a=1), (b=2), (c=1), (d=0), (a=1,b=2), (a=1,c=1), (a=1,d=0), (b=2,c=1), (b=1,d=0), (c=1,d=0), (a=1,b=2,c=1), (a=1,b=2,d=0), (a=1,c=1,d=0), (b=2,c=1,d-0), (a=1,b=2,c=1,d=0)} are tagged as n-FIC's and added to set S.

The result of test case 2 is also pass and hence all the t-way combinations {(a=1), (b=1), (c=0), (d=1), (a=1,b=1), (a=1,c=0), (a=1,d=1), (b=1,c=0), (b=1,d=1), (c=0,d=1), (a=1,b=1,c=0), (a=1,b=1,d=1), (a=1,c=0,d=1), (b=1,c=0,d=1), (a=1,b=1,c=0,d=1)} are tagged as n-FIC's and added to set S.

The result of test case 3 is also pass and hence all the t-way combinations{(a=2), (b=3), (c=1), (d=0), (a=2,b=3), (a=2,c=1), (a=2,d=0), (b=3,c=1), (b=3,d=0), (c=1,d=0), (a=2,b=3,c=1), (a=2,b=3,d=0), (a=2,c=1,d=0), (b=3,c=1,d=0), (a=2,b=3,c=1,d=0)} are tagged as n-FIC's and added to set S.

C. Step 3-Pinning down the FIC's from LFIC's:

In this step, we pin down the FIC's from the suspicious list. To identify the FIC from the suspicious list we generate additional test cases for each combination in the list as follows.

For each LFIC in the suspicious list, an additional test f is generated such that f contains the LFIC and the probability that f will fail is minimum. To find the probability we formulated the equation (1) similar to the one in BEN[9].

The process to generate the additional test case is as follows. First, generate a base test f as follows. For each parameter that is not involved in LFIC, choose a value whose suspiciousness value ρ is minimum. If there is more than one value with minimum ρ, one of them is selected randomly. Next, check whether the base test f is valid and has not been executed before. If so, f is returned as the new test that contains LFIC and has minimum ρ. If not, 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 valid and new test. These steps are repeated until a new, valid test is found.

$$\rho(o) = \tfrac{1}{3}\left(u(o) + v(o) + w(o)\right) \qquad \textbf{(1)}$$

Where

$$u(o) = \frac{\textit{number of failed test cases o has appeared}}{\textit{total number of failed test cases}}$$

$$v(o) = \frac{\textit{number of failed test cases o has appeared}}{\textit{total number of test cases o has appeared}}$$

$$w(o) = \frac{\textit{number of suspicious combinations with o}}{\textit{total number of suspicious combinations}}$$

We then execute the generated additional test case and if it passes, the corresponding LFIC is removed from the suspicious list and tagged as n-FIC. If it fails, the failure is more likely due to this LFIC since probability that f fails is minimized. Therefore, this LFIC is now tagged as FIC.

When a combination τ is marked as failure inducing, every combination s in suspicious list which contains LFIC is removed because the failure is caused by this LFICFor the example considered, the result of test case 4 is fail and the combinations {(a=1), (b=3), (c=1) , (d=0), (b=3,c=1), (a=1,c=1), (c=1,d=0), (a=1,d=0), (b=3,c=1), (a=1,c=1,d=0), (b=3,c=1,d=0)} are already marked as n-FIC's in S, therefore they cannot be the FIC's for the given test case. However, the combinations {(a=1,b=3), (a=1,b=3,c=1), (a=1,b=3,d=0) ,(a=1,b=3,c=1,d=0)} are not in S and hence tagged as LFIC's and added to the suspicious list.

For the combination (a=1,b=3) to generate additional test case we calculate ρ values for c=0,c=1 and d=0,d=1.

$$\rho(c=0) = \tfrac{1}{3}\left(\tfrac{0}{1} + \tfrac{0}{1} + \tfrac{0}{4}\right)$$
$$\rho(c=1) = \tfrac{1}{3}\left(\tfrac{1}{1} + \tfrac{0}{3} + \tfrac{2}{4}\right)$$
$$\rho(d=0) = \tfrac{1}{3}\left(\tfrac{1}{1} + \tfrac{1}{3} + \tfrac{2}{4}\right)$$
$$\rho(d=1) = \tfrac{1}{3}\left(\tfrac{0}{1} + \tfrac{0}{1} + \tfrac{0}{4}\right)$$
$$\rho(c=0) = \tfrac{1}{3}\left(\tfrac{0}{1} + \tfrac{0}{4} + \tfrac{0}{4}\right)$$

As $\rho(c=0)=0$ and $\rho(d=1)=0$

The new test case generated would be (a=1, b=3, c=0, d=1) and (a = 1, b = 3, c = 0, d = 1) fails. Therefore, the suspicious combination (a = 1, b = 3) is pinned down as a FIC. As all the combinations (a=1,b=3,c=1), (a=1,b=3,d=0) ,(a=1,b=3,c=1,d=0) contain this combination they are removed from the suspicious list and pinned down as a FIC. As (a=1, b=3) was a subset of all the likely failure-inducing combinations in the test case 4 it is identified as the FIC for the test case 4.

IV. Experiment

We implemented our algorithm in python and experimented on two programs. The first is the 'Account Program' taken from Software-Artifact Infrastructure Repository (SIR) [11] which deals with computing the balance in an account. The second is a web application for selling and buying cars obtained from internet source .

Account Program Case-study:

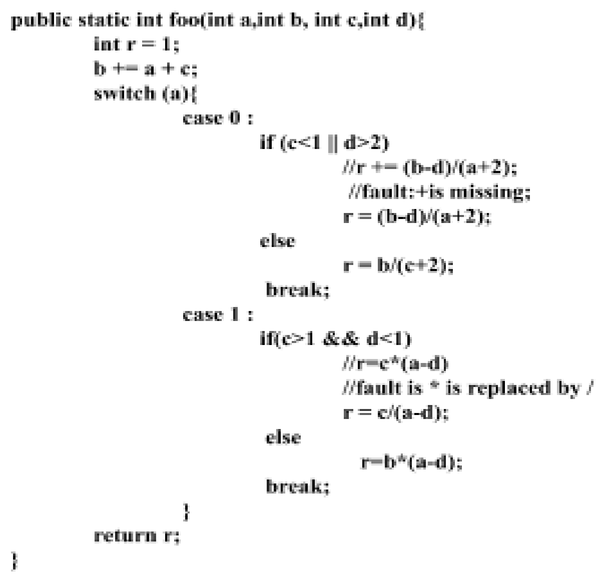
The account program is given in fig.2. The parameters to the program are P={a, b, c, d} and domain of the parameters are as follows d_a = {0,1}, d_b = {0,1}, d_c = {0,1,2}, and d_d = {0,1,2,3}.

Faults were seeded to the program such that the program would reach a faulty statements when:

- The value of a is 0 and the value of c is 0
- The value of a is 1,value c is 2 and value of d is 0
- The value if a is 0 and the value of d is 3

Car-sales Case- study:

The Car-sales is a web application for buying and selling cars. The web application has the parameters are P={order-category ,location, brand, Registration-number, Order-type} the values for the parameters are order-category={buy, sell} location={san-francisco, new-york} Brand={BMW, Mercedes, Audi} Registration-number={valid, invalid} order-type={store, online}



```
public static int foo(int a,int b, int c,int d){
        int r = 1;
        b += a + c;
        switch (a){
                case 0 :
                        if (c<1 || d>2)
                                //r += (b-d)/(a+2);
                                 //fault:+is missing;
                                r = (b-d)/(a+2);
                        else
                                r = b/(c+2);
                         break;
                case 1 :
                        if(c>1 && d<1)
                                //r=c*(a-d)
                                //fault is * is replaced by /
                                r = c/(a-d);
                        else
                                  r=b*(a-d);
                        break;
                }
        return r;
}
```

Fig 2.Account Program

The test case contains the values for parameters in P. Faults are seeded such that the results is faulty when:

- order-category='Buy' and location=='New-York'
- location='San-Francisco' and registration='Valid'
- order-category='Sell' and brand='Mercedes' and order-type='Store'

Test cases were generated by a tool ACTS[10]. The program was tested against the test cases using a test oracle. The original program without any faults was executed for all the test cases and this output was used by the test oracle to compare results and build the executed test file for the faulty program. This file was used to find the FICs using the tool. The results on these two case-studies are discussed in the next section.

TABLE-II FINDING FICS FOR ACCOUNT

Failed Test case				Suspicious Combinations	Additional test cases				Failure Inducing Combination
a	b	c	d		a	b	c	d	
0	0	0	0	('a', 'c'): ('0', '0'), ('a', 'b', 'c'): ('0', '0', '0'), ('a', 'c', 'd'): ('0', '0', '0'), ('a', 'b', 'c', 'd'): ('0', '0', '0', '0')	0	1	0	2	('a', 'c'): ('0', '0')
0	1	0	0	-	-	-	-	-	('a', 'c'): ('0', '0')
1	0	2	0	('a', 'c', 'd'): ('1', '2', '0'), ('a', 'b', 'c', 'd'): ('1', '0', '2', '0')	1	1	2	0	('a', 'c', 'd'): ('1', '2', '0')
1	1	2	0	-	-	-	-	-	('a', 'c', 'd'): ('1', '2', '0')
0	0	0	1	-	-	-	-	-	('a', 'c'): ('0', '0')
0	1	0	1	-	-	-	-	-	('a', 'c'): ('0', '0')
0	0	0	2	-	-	-	-	-	('a', 'c'): ('0', '0')
0	1	0	2	-	-	-	-	-	('a', 'c'): ('0', '0')
0	0	0	3	('a', 'd'): ('0', '3'), ('a', 'b', 'd'): ('0', '0', '3') ('a', 'b', 'c', 'd'): ('0', '0', '0', '3')	0	1	2	3	('a', 'd'): ('0', '3'), ('a', 'c'): ('0', '0')
0	1	0	3	-	-	-	-	-	('a', 'd'): ('0', '3'), ('a', 'c'): ('0', '0')
0	0	1	3	-	-	-	-	-	('a', 'd'): ('0', '3')
0	1	1	3	-	-	-	-	-	('a', 'd'): ('0', '3')
0	0	2	3	-	-	-	-	-	('a', 'd'): ('0', '3')
0	1	2	3	-	-	-	-	-	('a', 'd'): ('0', '3')

TABLE-III FINDING FICS FOR WEB APPLICATION

Failed Test case					# suspicious combinations	# additional test cases	Failure Inducing Combination
Order-category	Location	Brand	Reg No	Order Type			
Buy	N-Y	Merc	Valid	Online	11	1	(order-category, Brand):(Buy, Merc)
Buy	N-Y	Merc	Valid	Store	0	0	(order-category, Brand):(Buy, Merc)
Buy	N-Y	Merc	Invalid	Online	0	0	(order-category, Brand):(Buy, Merc)
Buy	S-F	Merc	Valid	Online	9	1	(Location, Reg No):(S-F, Valid)
Sell	N-Y	Merc	Valid	Store	7	2	(order-category, Brand, Order-Type): (Buy, Merc, Store)

V. Results and Discussion

The results for the 'Account' program are shown in TABLE-II. The column 1 shows the failed test cases for the program, the column 2 shows the suspicious combinations identified for each test case.
column 3 shows the additional test cases generated in each test case to find the FICs.
For the test case (0, 0, 0, 0) the suspicious combinations identified are ('a', 'c'): ('0', '0'),('a', 'b', 'c'): ('0', '0', '0'),
('a', 'c', 'd'): ('0', '0', '0'),('a', 'b', 'c', 'd'): ('0', '0', '0', '0').
For the combination ('a', 'c'):(0, 0) an additional test case is generated with value 'b'=1 and d='2'. This is the test case with the least possibility of failure. The new test case (0,1,0,2) is tested on the program and the result is 'Fail'. So the combination ('a', 'c'):(0, 0) is identified as a FIC. As the combination ('a', 'c'):(0, 0) is a subset of all the members in the suspicious set, the combination ('a', 'c'):(0, 0) is marked as the FIC for the test case (0,0,0,0).
For the test case (0, 1, 0, 0) as the combination ('a', 'c'):(0, 0) was identified as a FIC, the suspicious set is empty and ('a', 'c'):(0, 0) is identified as a FIC for the test case(0, 1, 0, 0). Similarly FICs were identified for each test case.
The results for web application are tabulated in table 2. For the test case ('Buy', 'New-York', 'Mercedes', 'Invalid', 'Online'). The suspicious combinations generated are ('Order-Category', 'Location'): ('Buy', 'New-York'), ('Order-Category', 'Location', 'Brand'): ('Buy', 'New-York', 'Audi'), ('Order-Category', 'Location', 'Reg-No'): ('Buy', 'New-York', 'Invalid'), ('Order-Category', 'Location', 'Order'): ('Buy', 'New-York', 'Online'), ('Order-Category', 'Location', 'Brand', 'Reg-No'): ('Buy', 'New-York', 'Audi', 'Invalid'), ('Order-Category', 'Location', 'Brand', 'Order'): ('Buy', 'New-York', 'Audi', 'Online'), ('Order-Category', 'Location', 'Reg-No', 'Order'): ('Buy', 'New-York', 'Invalid', 'Online'). Initially for the Suspicious combination ('Order-Category', 'Location'): ('Buy', 'New-York') a new test case is generated with brand=BMW ,Registration-no=invalid , order-type=online. This test case fails hence it the combination ('Order-Category',
'Location'): ('Buy', 'New-York') is marked as FIC. As all the members in the suspicious list contain ('Order-Category', 'Location'): ('Buy', 'New-York'), all of them are marked as FIC. Similarly FICs are identified for all the failed test cases as shown in table-III.

VI. Conclusion

In this paper, we work with an existing executed test suite as input to find the combinations of parameters that caused the fault to localize it. Our approach based on constraint satisfaction, automatically identifies the underlying FIC irrespective of whether it is a 2-way, 3-way or t-way.; whereas in BEN it has to be tried starting with 2-way and going all the way upto n-way for a specific failed test case till the FIC is uniquely found.
The approach has been tried on synthetic case-studies and found to be working well. In future, we plan to apply it on real-life test suites of projects in the industry.

References

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

[2] J. Li; C. Nie, and Y. Lei, "Improved Delta Debugging Based on Combinatorial Testing," In Proceedings of International Conference on Quality Software (QSIC), pp.102,105, 2012.

[3] A. Zeller and R. Hildebrandt. "Simplifying and isolating failure inducing input", In Proceedings of the IEEE Transactions on Software Engineering, 2002, pages 183–200.

[4] 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.

[5] 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.

[6] C. Nie, H. Leung, and B. Xu. "The minimal failure-causing schema of combinatorial testing", In ACM Transactions on Software Engineering and Methodology, 20(4) , September 2011.

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

[8] K. Shakya, T. Xie, N. Li, Y. Lei, R. Kacker, and R. Kuhn, "Isolating Failure-Inducing Combinations in Combinatorial Testing Using Test Augmentation and Classification," In proceedings of 5th IEEE International Conference on Software Testing, Verification and Validation (ICST), pp.620-623, 2012.

[9] Ghandehari, Laleh Sh, Yu Lei, Raghu Kacker, D. Richard Rick Kuhn, David Kung, and Tao Xie. "A Combinatorial Testing-Based Approach to Fault Localization." *IEEE Transactions on Software Engineering* (2018).

[10] Advanced Combinatorial Testing System (ACTS), http://csrc.nist.gov/groups/SNS/acts/documents/comparisonreport.html, 2015.

[11] Software-artifact Infrastructure Repository, http://sir.unl.edu/portal/index.php, 2012.