Isolating Failure-Inducing Combinations in Combinatorial Testing using Test Augmentation and Classification

Kiran Shakya   Tao Xie
North Carolina State University
{*kshakya,txie*}*@ncsu.edu*

Nuo Li
ABB Robotics
nuo.li@cn.abb.com

Yu Lei
University of Texas at Arlington
ylei@cse.uta.edu

Raghu Kacker   Richard Kuhn
Information Technology Lab, NIST
{*raghu.kacker,kuhn*}*@nist.gov*

Abstract—**Combinatorial Testing (CT) is a systematic way of sampling input parameters of the software under test (SUT). A t-way combinatorial test set can exercise all behaviors of the SUT caused by interactions between t input parameters or less. Although combinatorial testing can provide fault detection capability, it is often desirable to isolate the input combinations that cause failures. Isolating these failure-inducing combinations aids developers in understanding the causes of failures. Previous work directly uses classification tree analysis on the results of combinatorial testing to model the failure inducing combinations. But in many scenarios, the effectiveness of classification depends upon whether the analyzed test set is sufficient for classification. In addition, generating combinatorial tests for more-than-6-way combination is generally expensive. To address these issues, we propose an approach that uses existing combinatorial testing results to generate additional tests that enhance the effectiveness of classification. In addition, our approach also includes a technique to reduce the complexity of the resulting classification tree so that developers can understand the nature of failure-inducing combinations. We present the preliminary results of our approach applied on the TCAS benchmark.**

I. INTRODUCTION

A modern software system that is both large and complex has in general many parameters affecting the behavior of the overall system (such as configuration values). While validating the correctness of a software system across its entire input parameters is desirable, exhaustive testing of all combinations is not feasible. One practical and efficient way of dealing with this problem is Combinatorial Testing (CT) [1]. Given the input parameters that are properly modeled, t-way CT guarantees that a failure will be detected if this failure is caused by interaction among t parameters or less. Although CT can detect failures, it provides little support for diagnosing causes of failures. Specifically, during debugging, developers are interested in those specific combinations that induce the failures in the software under test (SUT). We call these combinations *failure-inducing combinations* or *faulty combinations*. Detecting these faulty combinations manually is generally a difficult task due to the large input space.

Previous work [2], [3], [4] on diagnosing the causes of failure has focused on using decision tree classification to characterize the faulty interactions. But much of the previous work focuses on applying classification after the tests have been generated and executed on the SUT. A common assumption made by the previous work is that the classification algorithm [5] performs well on combinatorial testing results. But in certain cases, the number of failing combinatorial tests might be very small, causing the classification algorithm to perform poorly. It is well known that decision tree classification can be biased if the dataset is highly unbalanced. Furthermore, developers may not know beforehand the number of parameters that can cause failures in the SUT. This kind of scenario is common when the SUT has a large number of parameters that affect its behavior and failures are caused by interaction among many parameters. In such cases, CT of higher strength can require more resources.

In order to address these challenges, we propose an approach that aids developers in situations where CT alone cannot enable effective failure diagnosis. We do not make any assumption regarding the size of faulty combinations and their natures (such as overlapping between faulty combinations). This characteristic makes our approach general and can be applied in many cases. The only assumption that we make is that the SUT is deterministic in nature (i.e., running the same test multiple times causes the execution of the SUT to produce the same test output).

In order to improve the effectiveness of classification, our approach includes a test augmentation technique that generates new tests using the available failing tests. Typically, the generated tests are mostly failing tests, thus improving the result of classification. In addition, our approach includes a feature selection technique that reduces the complexity of the resulting decision tree. We measure the complexity of the resulting decision tree in terms of the number of nodes in the tree. Developers use the decision tree not only to predict the fail or pass outcome of a test but also to analyze it manually for studying the nature of faulty interactions. Therefore, it is desirable to have a simple decision tree over a complex one.

This paper makes the following main contributions:

- We propose a test augmentation technique that aids a classification algorithm in situations where combinatorial test results are highly unbalanced.
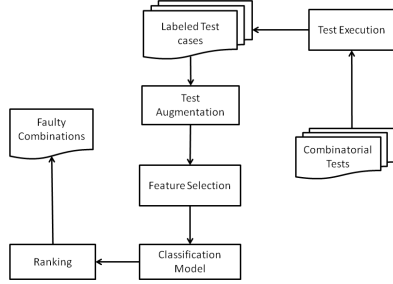
Figure 1: Overview of our approach

- We propose a feature selection technique to reduce the complexity of the resulting decision tree.
- We conduct a preliminary study to evaluate our approach.

II. BACKGROUND

Covering Array. If we assume that the SUT has k input parameters and each parameter c_i has a_i discrete possible values, the total number of possible inputs is $\prod_{i=1}^{i=k} a_i$. CT uses a covering array (CA) to systematically sample the large input space. A $CA(N, k, t, s)$ is an array of N rows and k columns, where N is the size of test set, k is the number of parameters, s is the number of possible values of each parameter, and t is the strength of CA. Given a strength t, for any t sub-columns of C, the sub-array covers all the possible combinations of the corresponding t parameters. Similarly, a mixed-level covering array (MCA) allows the parameters to have different number of possible values i.e. $s = \{v_1, v_2, ..., v_k\}$ where each ith parameter can take v_i distinct possible values.

Classification Tree. Classification tree [5] is a recursive partitioning approach to build models that predict class membership of an input. The input dataset is split according to the value of an attribute that maximizes the *Gain* in information. This splitting of the dataset continues until no further splitting is possible. There are many possible definitions for the *Gain*. We use a popular decision tree learning algorithm C4.5 [5], which defines the Gain function for feature c_i at node t as $Gain(c_i, t) = H(t) - H(c_i, t)$, where $H(t)$ denotes binary entropy at node t. If the prior probability of a parameter p taking certain value i is p_i, then the entropy for that parameter is given by $-\sum_i log_2 p_i$.

III. APPROACH

Figure 1 shows an overview of our approach. Our approach consists of four components: Test Augmentation, Feature Selection, Test Classification, and Ranking. We next present the details of each component.

A. Test Augmentation

Test Augmentation (TA) is used to improve the dataset when the classification algorithm performs poorly on the combinatorial test results. TA is particularly useful when the classification algorithm is unable to build a decision tree due to a small number of failing tests compared to passing tests. Given a failing test, TA uses the one factor one time (OFOT) [6] technique for generating additional test inputs. Specifically, if $t = (v_1, v_2, ..., v_k)$ is a test that fails, the TA generates new tests by replacing each v_i with another values of parameter c_i while keeping other parameter values the same. The rationale is that since the new test generated by the TA are similar to the failing test, these new test will also likely be failing tests, thus balancing the combinatorial test result set. We repeat this step for every failing test. In order to reduce the overhead, all the existing tests are kept in a cache, and the redundant tests generated by the TA (if any) are detected and discarded. If there are m failing tests, then the TA will generate at most $m \times \left(\sum_{i=1}^{k} a_i - k\right)$ tests that in general will be far less than the additional number of tests required to generate higher strength CT sets. The new tests are then executed and classified as failing or passing based on the execution.

B. Feature selection

Not all the parameters of the SUT correlate with the failures. If the SUT has a large number of parameters, it is necessary to prune the parameters (or features) in order to improve the effectiveness of the classification algorithm. Feature subset selection (FSS) is the process of identifying and removing irrelevant and redundant information as much as possible. This process reduces the dimensionality of the dataset and improves both speed and accuracy of the learning algorithm. There are a number of feature selection algorithms in the data mining literature [5]. In our current work, we have used correlation based feature selection (CFS) [7]. CFS takes into account the usefulness of individual parameters for predicting the class label along with the level of intercorrelation among them.

C. Test Classification

We use classification tree analysis (CTA) to model fault combinations. CTA classifies the labeled tests obtained from both the covering array and test augmentation. The classification is based on only the parameters selected by the feature selection component. The rationale is that classification based on selected features would result in a simpler decision tree which can be analyzed by the developers more easily. We use CTA because it has been used in previous work [2], [3], [4]. In order to reduce the overfitting problem commonly found in classification, we use n-fold cross validation [5]. Cross validation essentially builds multiple models from different subsets of input data and uses the results to identify the best model. We measure the effectiveness of the classification tree using F-score, which is a well known metric computed using two standard metrics:

$$Recall(R) = \frac{\# of\ correctly\ predicted\ failing\ tests}{total\ \#\ of\ failing\ tests}$$
$$Precision(P) = \frac{\#\ of\ correctly\ predicted\ failing\ tests}{total\ \#\ of\ predicted\ failing tests}$$

Table I: Characteristics of faults

Version	2-way	3-way	4-way	5-way
v16	0/156	1/461	6/1450	14/4309
v26	0/156	0/461	1/1450	18/4309

$$F\text{-}score = \frac{2PR}{P+R}$$

D. Ranking

Given a decision tree model, we enumerate all likely faulty combinations of the SUT. For each leaf node that indicates a failure, a corresponding likely faulty combination is computed by taking the conjunction of the parameter values found in the path from the root node to the leaf node. We then collect these faulty combinations and rank them before presenting to the developers. The rank of a faulty combination is determined by computing its F-ratio. If n is the total number of tests classified with a combination and m is the total number of failing tests correctly classified with this combination, then we define its F-ratio as m/n. The combinations are then sorted by their F-ratios in descending order and shown to the developers. We speculate that the actual faulty combinations are within top ten results.

IV. EVALUATION

In our evaluation, we have used a module of the Traffic Collision Avoidance System (TCAS) benchmark. TCAS has been used in other evaluations of testing approaches [8]. TCAS program takes 12 inputs and produces an output that can be either 0, 1, or 2. The total input combination of the TCAS is $3 \times 2^3 \times 3 \times 2 \times 4 \times 10^2 \times 3 \times 2 \times 3 = 1036800$, which is extremely large motivating the use of CT. Kuhn and Okun [8] generated a large number of combinatorial tests and produced corresponding testing results for 41 faulty versions of the TCAS program. For each faulty version, the testing result as failing or passing is determined by comparing the output of the faulty version with the correct version. We have used the same versions and test results in our evaluation.

Next, we present the preliminary results of our approach on two faulty versions of the TCAS program via version 16 and 26. From the results, we intend to answer following research questions:

1) **RQ1.** Does our test augmentation improve the classification?
2) **RQ2.** Does feature selection help in reducing the complexity of the resulting decision tree without compromising accuracy?
3) **RQ3.** Does our overall approach find the faulty combinations?

Table I shows the number of failing tests generated by CT from $t = 2$ way to $t = 5$ way tests for versions 16 and 26 of TCAS. We can see that the number of failing tests is very low compared to the number of passing tests.

In order to answer the research questions, we first ran the 5-way tests and generated an arff file, which is a format required by Weka [9]. Weka is an open source collection of algorithms for data mining tasks. We have used Weka's J48 tree classification component to generate the decision tree. Table II shows our evaluation results. We found that in both versions, Weka (with default settings) could not generate a decision tree model due to small number of failing tests. Next, we feed the arff files into our test augmentation component. Column "TA" shows the total number of new tests (only unique tests that did not exist in the current CT suite) generated as well as the number of tests that failed. We then added the new tests into the original data set and re-ran the classification algorithm. Column "F1" shows the F-score of the decision tree in addition to the precision and recall. The values for these metrics have been shown in format $Precision/Recall/F\text{-}Score$. This result shows that the test augmentation can improve the classification, answering our **RQ1**.

Table II: Evaluation Results

V	TA	F1	S1	FS	S2	F2
v16	302/357	.81/.67/.73	36/56	8	22/31	0.71/.59/.65
v26	407/407	.81/.79/.80	56/85	10	20/28	0.78/0.72/.74

Furthermore, we measured the complexity of the decision tree in terms of the number of leaves and nodes in it. Column "S1" shows the the complexity of the decision tree in format $\#leaves/\#nodes$. For example, for version 26, the total number of leaves was 56 and the total number of nodes was 85. Although the classification with the tree was effective, we found that size of the decision tree was too huge to be understood by the developers. We then ran the feature selection algorithm on the dataset. We used Weka's implementation of correlation-based feature selection algorithm (CfsSubsetEval) for feature selection. Column "FS" shows the number of attributes selected by the feature selection component. Next, we ran classification using only the attributes selected by the feature selection. Column "S2" shows the size of the resulting decision tree and its standard metric is shown in column "F2". For version 26, the result shows that the complexity of the decision tree is reduced from 85 to 28 nodes. Although the precision, recall, and F-score were a bit reduced by feature selection, the loss was not that significant. In addition, the decision tree was more compact and simpler than the previous decision tree answering our **RQ2**.

Finally, to answer **RQ3**, we used the final decision tree to enumerate the faulty combinations and rank the combinations according to their F-ratios. To verify the results, we manually analyzed the faulty versions of TCAS to find faulty combinations. We found it challenging to find the exact preconditions for failures due to the complexity of the program code. Therefore, we calculated only necessary conditions for the failures. For example, Figure 2 shows

```
tcas.c /* v26*/

1. int alt_sep_test(){
2.    ...
3. enabled = High_Confidence &&
4.    /*(Own_Tracked_Alt_Rate <= OLEV) &&  BUG */
5.    (Cur_Vertical_Sep > MAXALTDIFF);
6.    ...
```

Figure 2: Fault in TCAS version 26

the fault in version 26 of TCAS. Here the fault lies at Line 4 where the code is commented out. Therefore, the fault will cause a failure observed at the output only when `HighConfidence=1, OwnTrackedAltRate>OLEV(=600)` and `CurVerticalSep>MAXALTDIFF(=600)`. This combination was indeed within top 5 in the list generated by the ranking component. For version 16, our approach could catch one faulty combination out of two presented in the code.

V. RELATED WORK

Yilmaz et al. [4] used classification tree to analyze the results of CAs and detect potential faulty combinations for complex configuration spaces. They used the results of classification to build mixed-strength CAs to further enhance the efficiency of fault characterization. Fouché et al. [2] presented an improved algorithm for generating higher-strength arrays that reduces the cost and improves flexibility by reusing the tests from lower strength arrays. Mainly, it uses the results of lower strength covering array to generate higher strength covering array, saving the resources needed to run the entire tests. Dumlu et al. [3] conjectured that CAs may not test all t-combinations due to *masking effects*: failures that perturb execution so as to prevent other combination being exercised. They presented an approach to detect potential masking effects and then generate additional new CAs that allow previously masked combinations to be tested. They also use classification tree to isolate masking combination. Our work differs from these previous approaches because we a test augmentation and attribute selection techniques to improve failure classification.

Besides classification using decision tree, there exist other approaches that use some other techniques to detect the faulty combinations. Delta debugging [10] is an adaptive divide-and-conquer approach to locate faulty combinations. Similarly, Zhang et al. [11] also propose adaptive test generation that is similar to delta debugging and uses failing test as seed test to identify faulty combinations. In contrast, our approach is based on classification model rather than search-based techniques. Martínez et al. [12] define error locating arrays (ELAs) that can be used to locate faulty combinations between parameters under assumption that each parameter has safe values that do not participate the failures of the SUT. But finding these safe values may not be trivial for large-system. Ghandehari et al. [13] use ranking of sub-combinations in a combinatorial test set to identify potential faulty combinations and use the rank to further refine suspicious combinations. In contrast, our work is mainly based on ranking of classification results.

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented an approach that uses results of CT to isolate the faulty combinations that cause the SUT to fail. Specifically, we have proposed a general approach that can be applied to classify the faulty combinations in scenarios where failures are not commonly observed. Our approach includes two techniques, test augmentation and feature selection in the context of CT to enhance classification. We also presented some promising results of our approach.

In future work, we plan to infer the constraints among the faulty combinations rather than just enumerating them. For example, it is more insightful to report that the SUT fails whenever $c_1 > c_2 \wedge c_3 < 50$ instead of enumerating all combinations of c_1, c_2, and c_3 satisfying the constraint. Furthermore, we plan to evaluate our approach on larger subjects to assess the benefit of our approach.

VII. ACKNOWLEDGMENT

REFERENCES

[1] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, pp. 11:1–11:29, 2011.

[2] S. Fouché, M. B. Cohen, and A. Porter, "Incremental covering array failure characterization in large configuration spaces," ser. ISSTA, 2009, pp. 177–188.

[3] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter, "Feedback driven adaptive combinatorial testing," ser. ISSTA, 2011, pp. 243–253.

[4] C. Yilmaz, M. B. Cohen, and A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," in *ISSTA*, 2004, pp. 45–54.

[5] V. K. Pang-Ning Tan, Michael Steinbach, *Introduction to Data Mining*.   Addison-Wesley, 2006.

[6] C. Nie and H. Leung, "The minimal failure-causing schema of combinatorial testing," *ACM Trans. Softw. Eng. Methodol.*, pp. 15:1–15:38, 2011.

[7] H. A. Mark, "Correlation-based feature selection for machine learning," Ph.D. dissertation, Univ of Waikato, 1999.

[8] D. R. Kuhn and V. Okun, "Pseudo-exhaustive testing for software," in *SEW*, 2006, pp. 153–158.

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, 2009.

[10] A. Zeller, "Isolating cause-effect chains from computer programs," in *FSE*, 2002, pp. 1–10.

[11] Z. Zhang and J. Zhang, "Characterizing failure-causing parameter interactions by adaptive testing," in *ISSTA*, 2011, pp. 331–341.

[12] C. Martínez, L. Moura, D. Panario, and B. Stevens, "Algorithms to locate errors using covering arrays," ser. LATIN, 2008, pp. 504–519.

[13] L. S. G. Ghandehari, Y. Lei, T. Xie, D. R. Kuhn, and R. Kacker, "Identifying failure-inducing combinations in a combinatorial test set," in *ICST*, 2012.