Hata Verisi Analizi Ve Değişkenlerin Girdi Uzayının Ayrıklaştırılması Kullanan Bir Birleşimsel Test Verisi Üretimi Yaklaşımı

Hakan BOSNALI
ASELSAN, Ankara, Türkiye
Orta Doğu Teknik Üniversitesi, Ankara, Türkiye
hakanbosnali@gmail.com


Anahtar Kelimeler: Birleşimsel Test, Hata Verisi Analizi.
A Combinatorial Test Data Generation Approach Using Fault Data Analysis And Discretization Of Parameter Input Space

Hakan BOSNALI
ASELSAN, Ankara, Turkey
Middle East Technical University, Ankara, Turkey
hakanbosnali@gmail.com

Abstract. Combinatorial Testing is an efficient testing strategy. It is based on the idea that many faults are caused by interactions between a relatively small number of parameters. However, determining the right interaction strength to generate data for different software is an issue in terms of efficiency. In addition to that, it requires the inputs in a discrete form, while that is not always the case. We propose a new combinatorial test data generator tool that combines fault data analysis to determine the right interaction strength for the specific domain of software and transformation of the continuous input space of parameters into discrete using well known test techniques. With this new tool, it is aimed to minimize test costs, while maximizing the confidence in test data. Experiments made with the tool support this idea with results showing a significant increase in test efficiency.

Keywords: Combinatorial Testing, Fault Data Analysis.
1 Introduction

Generating efficient test data with limited resources has always been an issue for software testing. Most of the time, testing all possible combinations is impossible, since testing becomes exhaustive as the number of inputs increases. Among the numerous available methods, Combinatorial Testing is proven to be an effective testing strategy [1,3]. However, selecting the right degree of interaction is a problem and that affects the efficiency directly. For different software from different domains, the degree of interaction changes [1]. On the other hand, to be able to use this method, the range of inputs must be in a discrete form, but this is not the case at all times. Continuous ranges for numbers and uncountable combination of characters in a string field may not be suitable examples for inputs of this method. For these reasons, before using Combinatorial Testing, degree of interaction for the specific domain of software must be determined and the continuous input fields must be transformed into discrete.

In this paper, we aimed to develop an approach that overcomes these deficiencies. Since our working area is Black-box testing of software from Electronic Warfare and Radar domains, we had a valuable fault data for Graphical User Interface and Embedded software in these domains. Our purpose was analyzing the fault data to obtain the interaction strengths for each domain. Afterwards, we wanted to develop a combinatorial test data generator tool that first transforms the continuous input fields into discrete using well known test techniques, then generates combinatorial test data with the help of an existing tool and using the analysis results for the specified domain to select the degree of interaction. This way, the lacking parts in combinatorial testing method would be completed.

To validate this new tool and the method, we had to design experiments on a real industrial software from one of our domains to see if it helps reducing the effort and increasing the number of bugs found. We also wanted to compare the tests at the same setup with randomly generated test data to create a comparison baseline for the new approach. In these experiments, we tried to find out the duration of preparing test data and running the tests, the number of tests generated and the number of bugs found.

2 Background

Testing all possible combinations of parameters becomes exhaustive as the number of inputs increases. When there is a shortage of resources in cases like this, Combinatorial Testing method is very useful. It is a method that aims to increase the effectiveness of test procedure by reducing the number of test cases by selecting meaningful data according to a strategy. This method makes use of the idea that not every input parameter contributes to every fault generated by the system and many faults are caused by interactions between a relatively small number of parameters [1].

Pairwise Testing, which is a combinatorial method, suggest that many faults in the system are triggered by the interaction of two parameter values. Therefore, while generating input to the system, every combination of values of any two parameters must be covered by at least one test case.
Some empirical investigations have concluded that from 50 to 97 percent of software faults could be identified by pairwise combinatorial testing [1]. Therefore, in case of limited resources, Pairwise Testing may be a good way to find most of the faults in the system.

**T-way Testing**, which is the generalized form of Pairwise Testing, is based on the idea that some faults may be triggered by the interaction of more than two parameters. To generate these faults, interaction of t parameters is needed. Hence, in T-way testing, all combinations of any t parameters’ values must be included in at least one test case.

A 10-project empirical study [1] has shown that some faults were triggered by three, four, five, and six-way interactions in some systems including medical devices, browser, web server and NASA distributed database.

In the light of these studies, T-way testing is better than pairwise testing when more resources are available or more precise testing is required.

### 3 Related Work

The study [2] investigates the relation between the software fault types and the average t-way interaction that causes them. The terms “Bohrbugs” which are simple bugs and easy to reproduce, and “Mandelbugs” which are complex bugs and more difficult to reproduce are analyzed in terms of average interaction strength.

In this study, 242 bugs from bugs.mysql.com for MySQL software were examined. The results of the analysis on these data have shown that Mandelbugs have higher interaction strength than Bohrbugs. This means that more factors are needed to trigger more complex errors.

It is stated that the curves of two bug types are similar to each other but shifted by one factor. To be able to find all of bugs, 4-way testing for Bohrbugs and 5-way testing for Mandelbugs are sufficient.

To sum up, the idea that T-way testing of up to 6 factors provides the same level of confidence as exhaustive testing turned out to be true in MySQL case.

After analyzing the fault data, it is time to find a method to generate input data for the tests. A combinatorial design based approach for testing pairwise, or t-way combinations of system parameters is proposed in [3]. It takes the system parameters and relations between them as an input. System parameters must be defined in terms of possible values they have. However, these parameters may have some restrictions and constraints that affect the values of each other. For this reason, there is an interface for the user to keep out the disallowed tests. Then, using all these information, a combinatorial algorithm generates the test data according to the interaction strength determined by the user.

Some experiments are conducted on this approach. The results of the experiments with the AETG system demonstrate that it is an effective and robust method to produce input data for the tests.
Phases of Combinatorial Interaction Testing is explained in detail in [4]. Four phases consists of Modeling, Sampling, Test and Analyze. Modeling and Sampling phases define “What” to test, and Test and Analyze phases define “How” to test.

Modeling includes input space of SUT, configurations, constraints of system parameters etc. An input must be stated as a parameter that can have discrete values. If its possible values are continuous, then some common techniques like “Equivalence partitioning” and “Boundary Value Analysis” can be used to transform it to discrete. There may also be some constraints between the input parameters. To avoid invalid combinations of these parameters, constraints must be taken into consideration. Other than these, user may want to define some test data that must be included into test cases, or to remove unwanted or already tested cases. All of these forms the Modeling phase together. The problem with the Modeling is that generation, modification and maintenance of it must be done manually in most of the cases. It should be done in an automatic manner including constructing input space, generating constraints and configuration, and helping the decision of interaction strength.

4 Fault Data Analysis

In this part of the study, we gathered a valuable fault data from a total of 6 projects and 30 Computer Software Configuration Items (CSCI) in the Electronic Warfare and Radar domain. The projects consist of both embedded and Graphical User Interface software. Each project has on the average 3400 requirements and 584000 Lines of Code. In total, there were 1461 faults. We collected 10 features for each fault, which include Detailed explanation of the fault, Detection phase, Fault type, Severity, Reproducibility, Source of the fault, Solution type, Effort for solution, Verification method and Effort for verification.

We analyzed these data in order to design our future roadmap for improving our software test activities and making it more effective. While analyzing, we tried to find answer to certain questions. The question with the most remarkable answer was; What is the number of factors involved in failures?

First we categorized the projects according to their domain; Electronic Warfare (EW) and Radar, and software type; Embedded and GUI. After this categorization, we analyzed the “Detailed explanation of the fault” field to extract the interaction strength information for each fault. About 21% of the faults were not suitable for this analysis due to insufficient information given in the fault description.

Results were in harmony with the idea that 6-way combinatorial testing is effective as exhaustive testing [2]. In Fig. 1, number of interaction between parameters and corresponding cumulative error detection rate for all types of bugs are given. Radar Embedded bugs reach 100% coverage with an interaction strength of 4, Radar GUI bugs with 5, EW Embedded bugs with 5 and EW GUI bugs with 6. Hence, similarly in our case, 6-way combinatorial testing would be enough to find all the bugs.
After fault data analysis, we wanted to find a test data generation method using the results of the analysis. Since we had the interaction strength for each fault type and they were in compliance with the assumption “T-way testing of up to 6 factors provides the same level of confidence as exhaustive testing” made in [2], we decided to use combinatorial testing approach which is known to be an effective testing strategy [1,3].

First we investigated the combinatorial test data generator tools available. Most of the existing tools were built for pairwise testing [6] (e.g. IPO, AllPairs, Jenny). We needed a tool that is capable of generating test data up to 6-way interactions of input parameters according to our analysis results. Moreover, it had to be a desktop application that provides an API for external tools. FireEye t-way testing tool, later named as Advanced Combinatorial Testing System (ACTS), presented in [5] seemed to be suitable for these constraints and performed better than its counterparts [6]. It supports t-way test set generation with t ranging from 1 to 6. It provides a GUI and a command line interface for external usage to generate combinatorial test data with various options. Thanks to these features, it satisfies most of our restrictions.

Although ACTS answers most of our needs, it only accepts inputs in a discrete way. In our systems, we had continuous ranges for some parameters and this needed
to be handled first to be able to use ACTS. Therefore, we needed to develop a tool that completes the lacking abilities of ACTS for our test data development process.

5.1 Design Phase

We designed a tool that takes parameters and their possible values and transforms them into discrete values using well-known test techniques. Then, making use of the domain specific interaction strengths that we obtained during the analysis phase, it generates an input file for ACTS tool to generate combinatorial test data.

Our design structure consists of 3 stages: configuring the test data generation, discretizing the parameter values and generating the combinatorial test data using ACTS tool.

**Configuring the test data generation.** Our tool has two input files to configure the test data generation: one for the domain specific interaction strengths and one for the system parameters.

Interaction strength yields for the value “t” in t-way combinatorial testing. It also refers to “degree of interaction (doi)” between the parameters. In our tool, the interaction strengths for different domains can be configured easily with an xml file. We generated an xml file using the analysis results in Section 4 and added 4 domains and their interaction strength values. While selecting these values, we needed to make a trade-off between the number of test cases and the fault coverage because the number of test cases becomes larger as the interaction strength increases for full t-way coverage. For that reason, we analyzed the results in Section 4 and came up with the idea that the minimum interaction strengths that cover 90% or more of all faults would be optimum. According to this strategy, we generated the degree of interaction “doi” for each domain. This xml file can easily be extended for different domains just by adding a new line for the new domain and the degree of interaction value.

The second input file is for the system parameters, their values and test design techniques. It also contains the name of the system and the domain which the system belongs to. It can be in both xml and json format.

In the parameters section, for each parameter, there is a parameter name, resolution of its possible values, possible values and test design techniques that will be used in discretizing the continuous values. Possible values field may differ according to parameter type. Users may define a range, enter some specific values that must be included in the test data, and a string with a specified length. If the parameter is a string, then resolution field stands for the length of the string. Otherwise it implies the minimum amount of change inside the defined range. A parameter can have more than one technique to transform its continuous values to discrete.

Some combinations are not valid from the domain semantics, and must be excluded from the resulting test set. There is a constraints section inside the input file that allows users to specify conditions that will be taken into account during test data generation.
Discretizing the parameter values. Once the system, parameters and test techniques are created, the second step is discretizing the parameter values. In this step, we made use of the well known test design techniques such as Equivalence Partitioning and Boundary Value Analysis. We also added Error Guessing and String Analysis for the first version.

Equivalence Partitioning technique discretizes the continuous input by selecting a random value inside the given range of the parameter taking resolution into consideration. Boundary Value Analysis technique selects min and max values in the range of the parameter, also (max+resolution) and (min-resolution) values. Error Guessing technique just selects the values given in the input file. This technique was added to give the user the ability to add values that must be tested according to their experience. String Analysis technique is used to discretize the parameters of string type. It uses the resolution in the input file as the max length of the string, then selects a string with zero length, a random string with the length 1, and a random string with the max length.

After reading the input file, each parameter in the system transforms all continuous and undetermined values into specific, discrete and meaningful data using techniques defined for them.

Generating the combinatorial test data. Once we obtain the discrete parameter values, we need to feed these values to ACTS tool to generate the combinatorial test data. To be able to use ACTS tool, all system specifications, such as parameter values and constraints, should be transformed into an input file format which ACTS tool can understand. For the purpose of using ACTS tool, we generate a temporary input file with the information we have in our system.

After creating the input file for ACTS tool, we need to collect the domain specific degree of interaction value for the System Under Test (SUT). Our system parses the domain.xml file and retrieves the integer degree of interaction value for the domain which SUT belongs to. This value is used as the “t” value for t-way combinatorial test data generation.

Since we have the compatible input file and interaction strength for the parameters, we can run ACTS tool to generate test data. ACTS has a command line interface that helps users to exploit its features without GUI interaction. It takes the degree of interaction, the input file path and the output file path as command line parameters, then generates and writes the test data into the provided output file. We give the generated temporary input file path, degree of interaction value gathered for the SUT, and the output file path where the generated test data will be written.

With this final step, generation of t-way combinatorial test data for our system is completed.

6 An Experiment Using The Test Data Generator Tool

Once constructing the test data generator tool, we designed experiments to evaluate the tool to see if it really helps reducing test data generation effort and increasing the
number of bugs found. For this purpose, we chose a real industrial software in Radar Embedded domain, with a size of 7000 Lines of Code. This software was tested before by generating test data manually. It had more than 180 parameters and therefore many possible cases to be tested. Generating test cases for this software required a huge effort; that is why we chose it for subject program (SUT) in our experiment to see the effect of our tool on test effort.

One problem with the old test setup was setting parameter values manually, which required too much effort while our SUT had more than 180 parameters. Other than the effort of generating test data, another problem was selecting efficient test data with limited resources. In other words, since testing all possible cases was impossible, we needed to select the ones with higher chance to trigger bugs. For these reasons, our test data generator tool seemed to be a perfect fit for this case.

6.1 Test Environment Setup

We have an automatic test infrastructure which was developed in house for the purpose of testing software interfaces and certain test scenarios. It contains the interface descriptions and parameter information between different software. We can generate various test scenarios using this interface descriptions and set different parameter values. Then these scenarios are run automatically and expected test results are compared with the actual ones.

This test automation infrastructure uses an xml file that contains parameter names, their possible values or ranges, test techniques that will be used to analyze them, and constraints between these parameters. This xml file is created automatically from an interface description file that is used for developing the SUT. Then, our infrastructure uses this xml to create a test scenario.

First, we constructed a base test scenario using SUT’s interface descriptions. After that, we needed to generate test cases using our test data generator tool. However, the test scenario created by the infrastructure had to be converted to the input format of our tool. We wrote a simple adapter software for this purpose. It works as a translator between the test infrastructure and our test data generator tool to convert each tool’s output to the other one’s input. This way our test data generation cycle shown in Fig. 2, became fully automatic.

![Fig. 2. Test data generation cycle](image-url)
The adapter software converts the parameter information inside the base scenario into an input file for the test data generator tool. Then the tool generates the combinatorial test data for the selected domain using the related interaction strength which we obtain from the domain.xml file. First, the tool discretizes the possible values of the parameters according to test techniques and then generates the combinatorial test data according to constraints. After generating the test data, the adapter software generates the expected outcomes for input data with the help of interface descriptions and converts these into a test infrastructure scenario file to complete the cycle.

6.2 The Experiment and Test Results

Throughout the experiment, we tried to answer certain questions and obtain some useful metrics to compare the old and the new method. These questions were:

- How long did it take to prepare input files?
- How long did it take to carry out tests?
- How many tests were generated?
- How many bugs were triggered?
- How many new bugs were found?
- How many old bugs went unnoticed?

In the old method, we prepared input data and the expected outcomes manually with the help of our test infrastructure and ran them automatically. The process consisted of preparing one input and the related outcome at a time and running the test. Thanks to automatic test infrastructure, running one test took approximately 1 minute. However, preparing the input data and the expected outcomes took a huge effort. Taking that into account, preparing test cases and running the tests together cost 40 staff-hour in total. 190 test cases were generated manually for the tests and 11 bugs were found.

In the new method, we generated input data and expected outcomes with the test setup explained in Section 6.1. Preparing the base test scenario file took a little longer than the other steps in the test data generation cycle. After that, it only took 6 seconds to create test cases with our new setup. In total, generating test cases and running the tests cost only 8 staff-hour. Our combinatorial test data generator created 45 test cases and 15 bugs were triggered with this data set. Once we analyzed the results, we saw that 4 new bugs were discovered, and no old bugs went unnoticed with this new method.

When we analyzed newly found bugs, we saw that they were all triggered by the interaction between parameters. This showed that combinatorial testing strategy was effective for us to find new bugs. Two of these faults were exceptional cases where they could only be triggered by the interaction of two parameters when each value was at different partitions. Here, equivalence partitioning technique helped us find these bugs. Another fault was triggered by the interaction of two parameters when only one of them was outside the minimum boundary and the absolute value of it was smaller than the other parameter. Boundary value analysis technique helped us trigger
this fault. The last fault was caused when two parameters had string values with the length of zero. In this case, string analysis technique was effective.

The new method reduced the test effort by 80%. Moreover, although the number of test cases decreased by 76%, the number of bugs triggered increased by 36%.

To generate a comparison baseline for the new method, we also designed an experiment with randomly generated test data at the same test setup. For each parameter, we picked a random value in the range of the parameter for each test case and generated 45 test cases as in the new method. The results showed that only 7 faults were discovered where all the faults were caused by the value of one parameter and no interactions were needed to trigger them. When we compare this case with our new method, we see 114% increase in the number of bugs found.

6.3 Threats to Validity

The construct validity is related to possible issues in comparison baseline for our method. In the old method, tester’s bias while selecting test cases might have affected the results. To eliminate this threat, we designed another experiment with randomly generated test data in the same test setup. This way, the effect of tester was removed, and we had an objective test data and a more valid comparison baseline.

The internal validity is related to discretization techniques and test data generation algorithm in our approach. Different techniques and tools might affect test data and result in different behaviors. To overcome this threat, we used well known test techniques for the discretization of the parameter input space, and an existing tool (ACTS) to generate combinatorial test data. We selected these after reviewing all available techniques and tools in the literature.

The external validity is related to domain of software that is used in this study. We only applied our approach on Electronic Warfare and Radar domains, thus further analysis on software from different domains should be done for applicability.

The reliability states that this approach can be repeated by other researchers. For this purpose, the steps in Sections 4, 5 and 6 are explained in detail.

7 Conclusion

Software in Electronic Warfare and Radar domains have a large number of parameters with continuous input space. This situation results in uncountable possible combinations for test case generation. Since it is impossible to test all these combinations, we needed to find a method to minimize the number of test cases, while selecting ones with higher possibility of triggering bugs. Combinatorial Testing, which is proven to be an efficient testing strategy, seemed to be a good choice for our situation. However, it required inputs in a discrete form. Moreover, degree of interaction must have been different for each domain of software in terms of test efficiency issues.

The approach we proposed fills these gaps with fault data analysis to determine the right degree of interaction according to domain of software, and discretization of parameter input space using well known test techniques. Fault data analysis that we
made on different software domains helped us in specifying predefined degree of interaction values for each domain. These values were specified for future test data generation of specific domain of software. Discretization of continuous parameter input space was made using test techniques like Equivalence Partitioning, Boundary Value Analysis etc. This way, continuous range for a parameter was reduced to meaningful and discrete data with higher possibility to trigger bugs.

Our approach combined these methods with combinatorial testing to create an efficient test data generation method. Experiments showed that the new method reduced the test effort by 80%, the number of test cases decreased by 76%, and the number of bugs triggered increased by 36% compared to the old method. Moreover, it increased the number of bugs by 114% compared to tests with random data. In addition to these improvements, analysis of the new faults showed that test techniques and combinatorial approach directly played a role in finding them. These results indicates that this new approach is an effective way that minimize test costs, while maximizing the confidence in test data.

This approach was only applied to software from Electronic Warfare and Radar domains. However, since the approach is in a general form, it can be easily applied to software from other domains. This way, it can contribute to efficiency and reduce test data generation costs for software in all domains.

In the future, this approach can be extended to work with other test infrastructures as well. For this purpose, a generic API for external test automation tools can be provided.

References