A COMBINATORIAL TEST DATA GENERATION APPROACH USING FAULT DATA ANALYSIS AND DISCRETIZATION OF PARAMETER INPUT SPACE

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

HAKAN BOSNALI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
INFORMATION SYSTEMS

JUNE 2018

Approval of the thesis:

A COMBINATORIAL TEST DATA GENERATION APPROACH USING FAULT DATA ANALYSIS AND DISCRETIZATION OF PARAMETER INPUT SPACE

Submitted by **HAKAN BOSNALI** in partial fulfillment of the requirements for the degree of **Master of Science in Information Systems Department, Middle East Technical University** by,

Prof. Dr. Deniz Zeyrek Bozşahin _____
Dean, **Graduate School of Informatics**

Prof. Dr. Yasemin Yardımcı Çetin _____
Head of Department, **Information Systems**

Assoc. Prof. Dr. Aysu Betin Can _____
Supervisor, **Information Systems Dept., METU**

Examining Committee Members:

Prof. Dr. Halit Oğuztüzün _____
Computer Engineering Dept., METU

Assoc. Prof. Dr. Aysu Betin Can _____
Information Systems Dept., METU

Assoc. Prof. Dr. Erhan Eren _____
Information Systems Dept., METU

Assoc. Prof. Dr. Tuğba Taşkaya Temizel _____
Information Systems Dept., METU

Assist. Prof. Ayça Tarhan _____
Computer Engineering Dept., Hacettepe University

Date: 22.06.2018

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Hakan Bosnalı

Signature :

ABSTRACT

A COMBINATORIAL TEST DATA GENERATION APPROACH USING FAULT DATA ANALYSIS AND DISCRETIZATION OF PARAMETER INPUT SPACE

Bosnalı, Hakan
M.Sc., Department of Information Systems
Supervisor: Assoc. Prof. Dr. Aysu Betin Can

June 2018, 54 pages

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

ÖZ

HATA VERİSİ ANALİZİ VE DEĞİŞKENLERİN GİRDİ UZAYININ AYRIKLAŞTIRILMASINI KULLANAN BİR BİRLEŞİMSEL TEST VERİSİ ÜRETİMİ YAKLAŞIMI

Bosnalı, Hakan
Yüksek Lisans, Bilişim Sistemleri Bölümü
Tez Yöneticisi: Doç. Dr. Aysu Betin Can

Haziran 2018, 54 sayfa

Birleşimsel Test yöntemi verimli bir test stratejisidir. Bu yöntem, çoğu hatanın nispeten düşük sayıda değişkenin etkileşimi sonucu meydana geldiği kanısına dayanır. Fakat farklı yazılımlar için doğru etkileşim sayısını belirlemek verimlilik anlamında bir sorundur. Bununla birlikte, girdilerin ayrık bir biçimde olması gerekmektedir. Hata verisi analizi ile doğru etkileşim sayısını belirlemeyi ve değişkenlerin sürekli olan girdi uzayının iyi bilinen test teknikleri kullanılarak ayrıklaştırılmasını bir araya getiren yeni bir birleşimsel test verisi üreteci öneriyoruz. Bu yeni araçla, test maliyetleri en aza indirilirken, test verisine olan güvenin de en üst düzeye çıkarılması hedeflenmiştir. Yeni araçla yapılan deneyler de, sonuçlarda görülen test verimliliğinin önemli ölçüde artış göstermesi ile bu fikri desteklemektedir.

Anahtar Kelimeler: Birleşimsel Test, Hata Verisi Analizi

To My Family

ACKNOWLEDGEMENTS

I wish to express my deepest gratitude to my supervisor Doç. Dr. Aysu Betin Can for her guidance, advice, criticism, encouragements and insight throughout the research.

I would also like to thank my colleague Ms. Elifnur Öztürk for her support in experiments.

Besides, I would like to express my appreciation to my mother Emine Baybaş, my father Bülent Bosnalı and my sister Gözde Bosnalı for always supporting me.

TABLE OF CONTENTS

ABSTRACT V
ÖZ VI
ACKNOWLEDGEMENTS VIII
TABLE OF CONTENTS IX
LIST OF TABLES X
LIST OF FIGURES XI
LIST OF ABBREVIATIONS XII
CHAPTER 1 1
INTRODUCTION 1
CHAPTER 2 3
BACKGROUND 3
2.1. Test Techniques 3
2.1.1. Equivalence Partitioning 3
2.1.2. Boundary Value Analysis 3
2.1.3. Error Guessing 4
2.2. Combinatorial Testing 4
CHAPTER 3 9
LITERATURE REVIEW 9
3.1. Fault Taxonomy 9
3.2. Fault Diagnosis Ontology 12
3.3. Software Fault Categorization 13
3.4. Trends in Software Fault and Failure Data 17
3.5. Combinatorial Testing 19
CHAPTER 4 23
FAULT DATA ANALYSIS 23
4.1. Distribution of the faults according to detection phase 25
4.2. Distribution of the faults according to fault type 26
4.3. Distribution of the faults according to source of the fault 27
4.4. The average effort for solution according to source of the fault 28
4.5. The average effort for solution of the most critical faults according to detection phase 29
4.6. The number of factors involved in failures 30
CHAPTER 5 33
COMBINATORIAL TEST DATA GENERATOR 33
5.1. Design Phase 34
5.1.1. Configuring the test data generation 36
5.1.2. Discretizing the parameter values 40
5.1.3. Generating the combinatorial test data 41
CHAPTER 6 45
AN EXPERIMENT USING THE TEST DATA GENERATOR TOOL 45
6.1. Test Environment Setup 45
6.2. The Experiment and Test Results 46
6.3. Threats to Validity 50
CHAPTER 7 51
CONCLUSION 51
REFERENCES 53

LIST OF TABLES

Table 1. Pairwise test set for 3 Boolean inputs 5
Table 2. An illustration of a system with 4 parameters 6
Table 3. 3-way test set for the system 7
Table 4. The error taxonomy presented by [2] 10
Table 5. Average size metrics per CSCI according to domain and software type 23
Table 6. Fields of faults and their possible values 24

LIST OF FIGURES

Figure 1. Cumulative error detection rate vs number of interaction between parameters [1] 8
Figure 2. Taxonomy of faults, combined with observed effects [4] 11
Figure 3. Overview of the Fault Diagnosis Ontology framework [5] 13
Figure 4. Pareto diagram for GCC [12] 18
Figure 5. Sources of failures for NASA mission [12] 18
Figure 6. Interaction strength and Average interaction strength for MySQL bugs [13] 20
Figure 7. Four phases of CIT [15] 21
Figure 8. Distribution of the faults according to detection phase 26
Figure 9. Distribution of the faults according to fault type 27
Figure 10. Distribution of the faults according to source of the fault 28
Figure 11. The average effort for solution according to source of the fault 29
Figure 12. The average effort for solution of the most critical faults according to detection phase 30
Figure 13. Interaction strength for EW and Radar bugs 31
Figure 14. Average interaction strength for EW and Radar bugs 32
Figure 15. Algorithm IPOG-Test [19] 34
Figure 16. Class Diagram for Test Data Generator tool 35
Figure 17. Content of domain.xml file 36
Figure 18. Content of a json input file 38
Figure 19. Content of an xml input file 39
Figure 20. Sequence diagram for discretizing the parameter values 41
Figure 21. Content of an ACTS input file 42
Figure 22. Sequence diagram for generating the combinatorial test data 44
Figure 23. Test data generation cycle 46
Figure 24. Comparison of effort in staff-hour 48
Figure 25. Comparison of number of test cases 48
Figure 26. Comparison of number of bugs found between old method and new method 49
Figure 27. Comparison of number of bugs found between tests with random data and new method 50

LIST OF ABBREVIATIONS

CSCI	Computer Software Configuration Item
GUI	Graphical User Interface
EW	Electronic Warfare
ACTS	Advanced Combinatorial Testing System
IPOG	In-Parameter-Order-General
SUT	System Under Test

CHAPTER 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] [14]. 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 this approach, degree of interaction for the specific domain of software must be determined and the continuous input fields must be transformed into discrete.

In this thesis, 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. We analyzed these fault data to obtain the interaction strengths for each domain. Afterwards, we developed 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 using the analysis results for the specified domain to select the degree of interaction. This way, the lacking parts in combinatorial testing method were completed.

To validate this new tool and the method, we designed experiments on a real industrial software from Radar Embedded domain to see if it helps reducing the effort and increasing the number of bugs found. The results showed that test effort was reduced by 80%, the number of test cases decreased by 76% and the number of bugs triggered increased by 36% comparing with the old method. Analysis of new

bugs showed that they were all triggered by the interaction between parameters. Moreover, test techniques used to discretize parameter input space also helped generating test cases to trigger exceptional faults.

We also re-run the tests at the same setup with randomly generated test data to create a comparison baseline for the new approach. Comparing the results of new method with this case, the number of bugs found increased by 114%.

In this approach, we combined fault data analysis to determine the right interaction strength according to domain of software, and discretization of continuous input space of parameters using well known test techniques to generate combinatorial test data. With this new tool, we aim 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.

The rest of the thesis is organized as follows. Chapter 2 gives an insight about the terms and methods that are used in this study. Chapter 3 consists of the review of the similar studies in the literature. Chapter 4 includes the fault data analysis of software in our domains. The design and work flow of test data generator tool is explained in detail in Chapter 5. Experiments on the tool and the results are given in Chapter 6. Finally, Chapter 7 contains conclusion and future work.

CHAPTER 2

BACKGROUND

2.1. Test Techniques

In this section, we give related background information about software testing techniques, especially black box techniques as they are in our scope.

Black Box Testing, is a software testing method that examines the functionality of a software without knowing its internal structure. It is interested in "What" the software is supposed to do, not "How" the software does it. It treats the system as a black-box and analyzes what outputs the system gives according to specific inputs. Test cases are derived from external descriptions of the software, including specifications, requirements and design parameters. Correct output is determined, often with the help of an oracle or a previous result that is known to be good. Below we list a sample of black box techniques.

2.1.1. Equivalence Partitioning

In this technique, inputs that result in similar behaviors of the system are divided into groups, so they are treated the same way. Test cases are designed to cover each partition at least once. For both valid and invalid data, equivalence partitions can be formed. Partitions can also be generated for outputs, internal values, time related values and interface parameters. It can be applied at any phase of testing.

Equivalence partitioning aims to achieve input and output coverage. It reduces the time required for testing a software by decreasing the number of test cases.

2.1.2. Boundary Value Analysis

The edge values of an equivalence partition are more likely to behave different than the values within the partition. Therefore, testing the boundaries is likely to find some defects for the system. The maximum and the minimum values of a partition are called its boundary values. Both valid and invalid boundaries can be chosen while designing test cases. In this case, test vectors consist of the values which are on the either side of the boundary.

Boundary value analysis can be applied at any level of testing. Its application is easy and it is capable of finding defects with high probability.

2.1.3. Error Guessing

Error guessing is a commonly used experience-based technique. Testers can foresee some defects based on their experiences. Therefore an approach can be constructed by listing possible defects and designing test cases to find these defects. This approach is called as fault attack. The lists for the defects and failures can be formed based on experience, data available and from common knowledge.

2.2. Combinatorial Testing

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. For example, let us think of a system with three Boolean inputs. All possible combinations are $2^3 = 8$ tests. In pairwise testing method, we can cover all pairs with only 4 tests as seen in Table 1. Boolean values are denoted as 0 and 1. When you check each parameter pair, Parameter 1 and 2, Parameter 1 and 3, and Parameter 2 and 3, each set contains all possible combinations of two Boolean inputs {00, 01, 10, 11}. In this system, all the faults that are triggered by the interaction of at most two parameters as indicated in the Pairwise Testing method, can be found with only 4 test cases.

Table 1. Pairwise test set for 3 Boolean inputs

	Parameter 1	**Parameter 2**	**Parameter 3**
Test case 1	0	0	0
Test case 2	0	1	1
Test case 3	1	0	1
Test case 4	1	1	0

In this example, the difference between the test cases of two methods is only 4. However, as the number of input parameters and the possible values of them increase, the gap between the number of the test cases and the effort needed to apply them increase drastically. To illustrate, let a system consist of 4 controls, each having respectively 2, 3, 4 and 7 possible settings. There must be $2\times3\times4\times7 = 168$ test cases to cover all possible combinations. In this case, we can cover all pairs with only 28 test cases, which is a quite logical choice for the sake of effectiveness.

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.

For instance, let us consider the system in Table 2. It has 4 parameters with possible values of 2, 5, 3 and 2 respectively. To cover all the possible combinations of these parameters, we need $2\times5\times3\times2 = 60$ test cases.

Table 2. An illustration of a system with 4 parameters

Parameter name	Parameter Value
a	[true, false]
b	[0, 1, 2, 3, 4]
c	[qwert, zxcv, 265rge]
d	[ghn, 78ok]

Now let us think we do not have enough resources to test all the combinations and decide to prepare a test set to cover only up to 3-way interactions. In this manner, a test set in Table 3 is prepared. As seen, only 30 test cases are enough for 3-way interactions of all parameters. When any 3 columns are picked, it can be seen that all the combinations of values for these 3 parameters are covered.

Table 3. 3-way test set for the system

Test Case No.	**a**	**b**	**c**	**d**
1	true	0	qwert	ghn
2	false	0	qwert	78ok
3	true	0	zxcv	78ok
4	false	0	zxcv	ghn
5	true	0	265rge	ghn
6	false	0	265rge	78ok
7	true	1	qwert	78ok
8	false	1	qwert	ghn
9	true	1	zxcv	ghn
10	false	1	zxcv	78ok
11	true	1	265rge	78ok
12	false	1	265rge	ghn
13	true	2	qwert	ghn
14	false	2	qwert	78ok
15	true	2	zxcv	78ok
16	false	2	zxcv	ghn
17	true	2	265rge	ghn
18	false	2	265rge	78ok
19	true	3	qwert	ghn
20	false	3	qwert	78ok
21	true	3	zxcv	78ok
22	false	3	zxcv	ghn
23	true	3	265rge	ghn
24	false	3	265rge	78ok
25	true	4	qwert	ghn
26	false	4	qwert	78ok
27	true	4	zxcv	78ok
28	false	4	zxcv	ghn
29	true	4	265rge	ghn
30	false	4	265rge	78ok

A 10-project empirical study [1] has shown that some faults were triggered by three-, four-, five, and six -way interactions in some systems which can be seen in Figure 1.

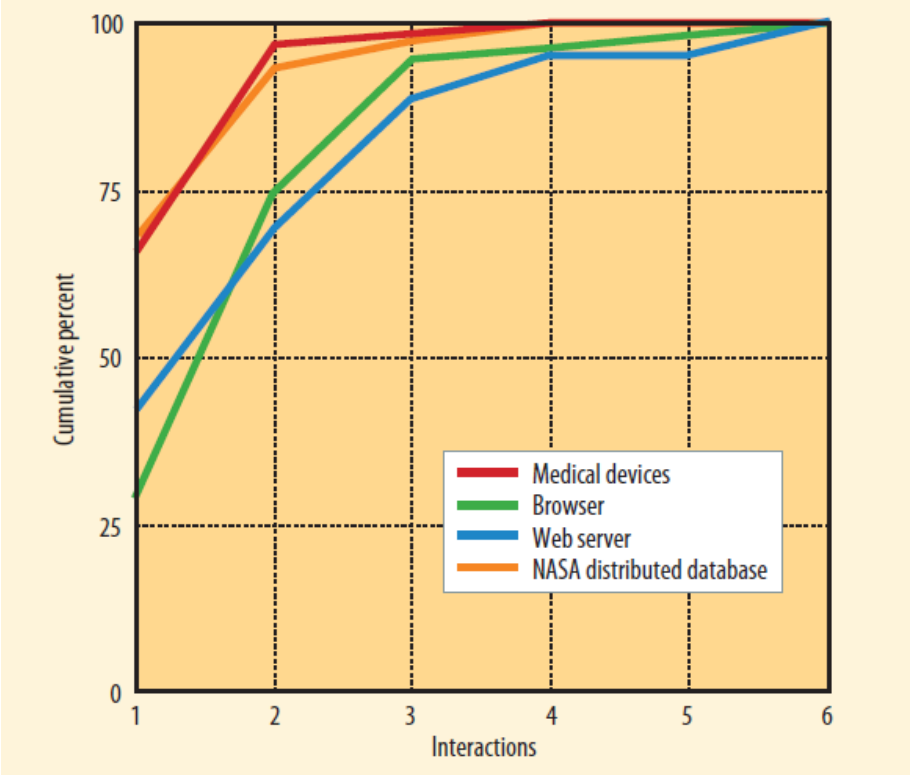

Figure 1. Cumulative error detection rate vs number of interaction between parameters [1]

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.

CHAPTER 3

LITERATURE REVIEW

Software fault data is very important in terms of analyzing similar systems to prevent future occurrence of similar faults. Many approaches are proposed in order to achieve this goal, such as fault categorization, fault prediction, ontology based fault diagnosis and fault taxonomies etc. In all of these approaches, recorded fault data from earlier systems is in the center of all structure. Once the data is provided, the next step is to find the suitable method to analyze it.

3.1. Fault Taxonomy

There are a number of studies on fault taxonomy. Here we discuss fault taxonomy studies for component-based software, service oriented architecture and web service composition.

In [2], a software fault taxonomy for component based software is proposed. Known faults are classified regarding their causes and effects. In this model, there are mainly two fault classes: service-related and structure-related faults. Service-related faults are divided into subgroups as syntactic, semantic, or non-functional. On the other hand structure-related faults, which are related to the structure of the system, are grouped according to causes of faults such as faulty connectors, the infrastructure, and the topology. In Table 4, all main categories and sub-categories of them can be seen. As a result, it is stated that, classification of the faults in this manner is the initiative for developing effective tests and analysis, and identification of common faults.

Table 4. The error taxonomy presented by [2]

Main Category	**Sub-Categories**
Syntactic	• Interface Violation
Semantic	• Misunderstood on the Behavior • Misunderstood on Parameters • Misunderstood on Events • Misunderstood on the Interaction Protocol
Non-Functional	• Performances • Quality of Service
Connectors	• Disagreement on the Protocol • Quality of Service
Infrastructure	• Underlying Services • Underlying System
Topology	• Callback • Re-entrance • Recursion
Other	• Multi-thread • Heterogeneous Languages • Persistence • Inconsistence Events

Another taxonomy is presented for Service-Oriented Architecture in [3]. This approach makes use of the steps of Service-Oriented Architecture to distinguish fault types, which are publishing, discovery, composition, binding, and execution. Publishing faults are said to occur when the system description is incorrect or it does not match the deployed service. Discovery faults can happen because of a non-existing service or not being listed in lookup service. Composition faults can take place in case of incompatible components. Binding faults may arise due to authorization, authentication or accounting problems. When there is a mismatch between the result and the expected outcome, execution faults occur. In conclusion, it

is said that this distinction is important for building dependable systems and testing the system via fault injection.

The study [4] proposes a fault taxonomy for web service composition. The faults are first divided into three major groups according to the causes: Physical, Development and Interaction faults. Then starting from these top-level groups, subcategories are formed and 6 fault classes are listed: development, operational, internal, external, hardware and software faults. Finally, with the help of observed effects, a matrix in Figure 2 is constructed. This approach is used for developing recovery techniques from failures.

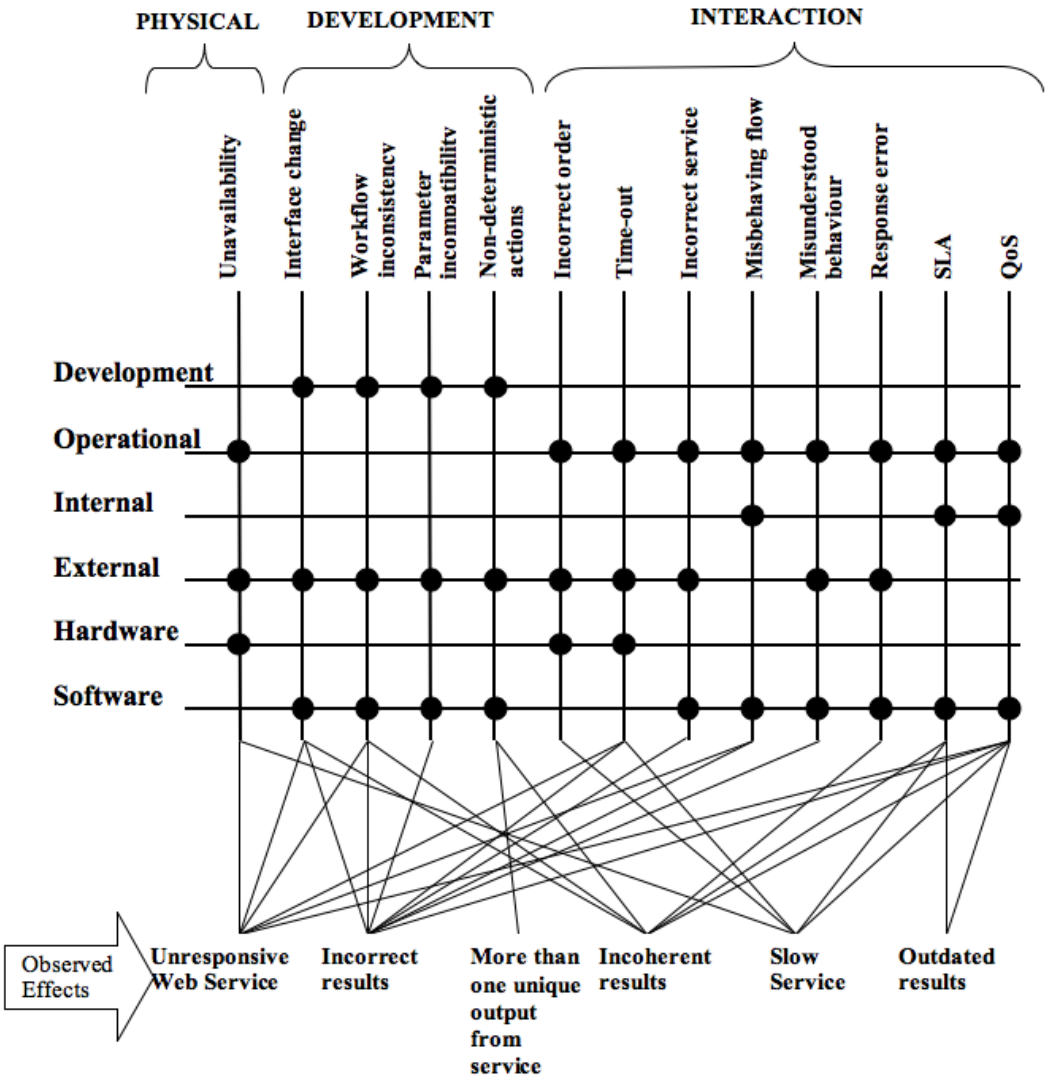

Figure 2. Taxonomy of faults, combined with observed effects [4]

3.2. Fault Diagnosis Ontology

The study [5] proposes an ontology-based software test generation framework. It is used in automated test case selection as an application of knowledge engineering. The framework has four phases as seen in Figure 3. First phase is Test Objective Generation, which is managed by ontology. It takes rules, expert knowledge and behavioral model as input and generates test objectives. Second phase is Redundancy Checking, which checks test cases for the sake of fulfilling the objectives. Test objectives and some rule templates are fed into this phase and non-redundant test objectives are formed. In the third phase called Abstract Test Suite Ontology Generation, abstract test cases are generated using test generation methods in literature. Last phase is Executable Test Suite Generation. In this phase, executable test cases are generated from abstract test suite. In overall, this method depends on the solidity of rules and ontology given to the system. It is flexible and can be extended with the introduction of new behavioral models and expert knowledge.

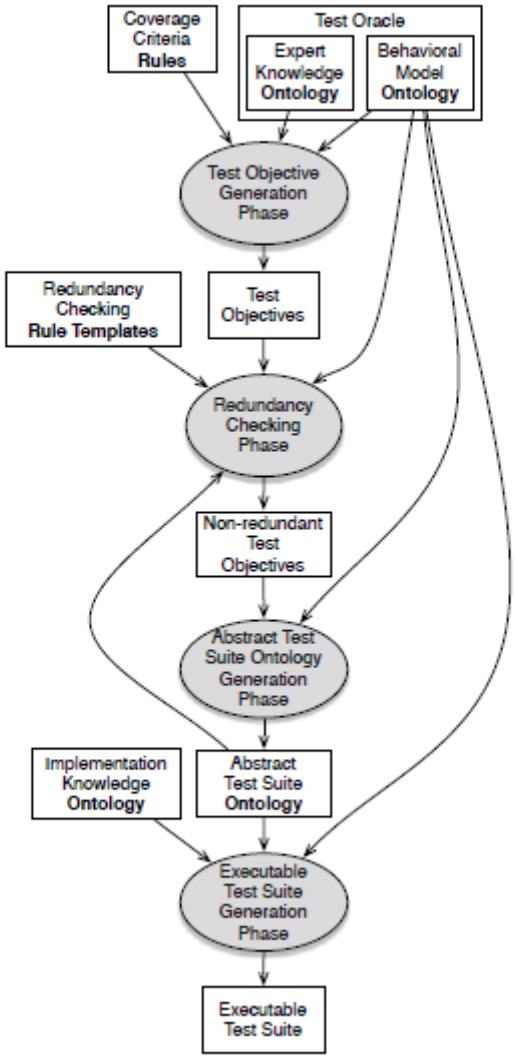

Figure 3. Overview of the Fault Diagnosis Ontology framework [5]

3.3. Software Fault Categorization

In [6], some important studies on software fault categorization in the literature are analyzed. It summarizes these studies stating the fault categories and the fault attributes that help them build these categorization.

The first study is Knuth's errors of TEX [7]. It is based on the data collected from software projects for 10 years. Knuth has listed 9 fault classes each denoted by a letter:

- "*A algorithm awry*
- *B blunder or botch*

- *D data structure debacle*
- *F forgotten function*
- *L language liability*
- *M mismatch between modules*
- *S surprising scenario*
- *T trivial typo*" [7]

The below attributes of faults are used to construct these categories.

- "*Time of fault introduction (e.g., A - during specification of an algorithm, B - during coding)*
- *Fault location (e.g., D - data structure; faults are also assigned to application modules)*
- *Manifestation in source code or lack thereof (errors of commission and omission, e.g. F - forgotten function)*
- *Sort of information misinterpreted by the programmer (e.g., M - module interfaces, L - language rules)*
- *Programmer's ability to avoid the fault at the time of its introduction (e.g., B - easily avoidable blunder, S - difficult-to-predict future scenario)*" [7]

Performance optimizations are not included in Knuth's categorization. The fault frequencies could have also been included as an attribute.

The second study is Beizer's bug taxonomy [8]. Its data was collected from many software systems including defense, aerospace and communication domains and written in different programming languages. A total of 982 bugs were used to develop this classification. The attributes used to classify faults are:

- "*Time of fault introduction (e.g., specification, implementation, test)*
- *Effects of fault activation (e.g., undesired control flow, data corruption)*
- *Location (e.g., the entity which is deemed incorrect and requires fixing)*
- *Type of required corrective action (e.g., requirement wrong/undesirable/not needed/ambiguous, data scope: "local should be global")*" [8]

Using these attributes, a hierarchical tree was formed with 8 top level categories and over 100 leaf categories. Top level categories are:

- "*Functional bugs: requirements and features (errors during specification of requirements)*
- *Functionality as implemented (errors in the interpretation of requirements)*
- *Structural bugs (mistakes during implementation concerning control flow and expression manipulation)*
- *Data bugs (bugs in definition, structure or use of data)*
- *Implementation (typographical bugs, violations of standards and conventions, errors in documentation)*
- *Integration (bugs having to do with interfaces between components)*
- *System and software architecture*
- *Test definition and execution*" [8]

Frequency for each category is also presented in this study.

The third study is Gray's classification of software faults in production software [9]. Gray did not propose a fault categorization, but analyzed the reasons of failures. In this manner, the faults are divided in two types:

- "*Transient faults (called Heisenbugs in the original paper) whose activations can be masked by restoring a consistent initial state and retry; in other words they cannot be simply reproduced by repeated execution.*
- *Non-transient faults (called Bohrbugs) whose activations cannot be masked by retry; these are the easily reproducible faults.*" [9]

Unlike others, Gray categorizes software faults with respect to their runtime behavior.

Orthogonal Defect Classification [10] is the fourth study. It was produced by analyzing over 50 software projects at IBM. Defect classes found in this study are:

- "*Function (missing or wrong functionality, may require a formal design change)*
- *Interface (addresses errors in communication between users, modules or device drivers)*
- *Checking (faulty or missing validation of data and values in the source code)*
- *Assignment (addresses faults in the source code such as faulty initialization)*

- *Timing/Serialization (errors that are corrected by improved management of shared and real-time resources)*
- *Build/Package/Merge (addresses problems due to mistakes in library systems, management of changes, and version control)*
- *Documentation (addresses publications and maintenance notes)*
- *Algorithm (addresses efficiency or correctness problems which can be fixed by re-implementation without the need for requesting a design change)*" [10]

An attribute called "defect trigger" is specified to address the activation of faults.

Another study analyzed in this paper is Eisenstadt's bug war stories [11]. He asked many software developers about their most remarkable bug hunting stories. There were three questions to answer: "why difficult", "how found", and "root cause". Third question yields a schema for fault categorization:

- "*mem – Memory clobbered or used up*
- *vendor – Vendor's problem (hardware or software)*
- *des.logic – Unanticipated case (faulty design logic)*
- *init – Wrong initialization; wrong type; definition clash*
- *lex – Lexical problem, bad parse, or ambiguous syntax*
- *var – Wrong variable or operator*
- *unsolved – Unknown and still unsolved to this day*
- *lang – Semantics ambiguous or misunderstood*
- *behav – End user's (or programmer's) subtle behavior*
- *??? (No information)*" [11]

This study is based on the best practices of developers and aims to guide design of the future debugging tools.

The last study is "A grammar based fault classification schema by DeMillo and Mathur" [11]. It is suggested that categorization of frequent faults can be used to select suitable test techniques. A grammar based fault categorization with the following categories is proposed:

- "*Spurious entity – requires the removal of its characteristic substring*
- *Missing entity – requires the insertion of a syntactic entity*
- *Misplaced entity – requires a change in its position*

- *Incorrect entity – for all other faults*" [11]

This approach is said to be appropriate for procedural programming languages, and may not be applicable to functional or logic programming languages.

In conclusion, it is stated that project specific factors such as maturity of the software, operating environment or programming language have a quite big effect on software faults. Another information given at the end is that a systematic approach for handling these factors has not been established.

3.4. Trends in Software Fault and Failure Data

The paper [12] analyzes fault and failure data from two large, real world case studies which are the open-source application GCC, and a large-scale NASA mission. Mainly two questions are asked and used as a guide while analyzing these data:

- "*Are faults that cause individual failures localized, that is, do they belong to the same file, component, top level component, etc.?*
- *Are some sources of failures (i.e., types of faults) more common than others?*" [12]

Some other software terms that are taken into consideration in this paper are:

- "*The Pareto principle (i.e., a small number of modules contain the majority of faults)*
- *Fault persistence through the testing phase and pre and post release*
- *The relationship between lines of code and faults*
- *Similarities in fault densities within project phases and across projects.*" [12]

Investigation of Pareto principle on GCC, which can be seen in Figure 4, have shown that 20% of the files contain nearly 80% of the faults. Moreover, 100% of the faults are found in only 47% of the files. This result is perfectly parallel with Pareto principle.

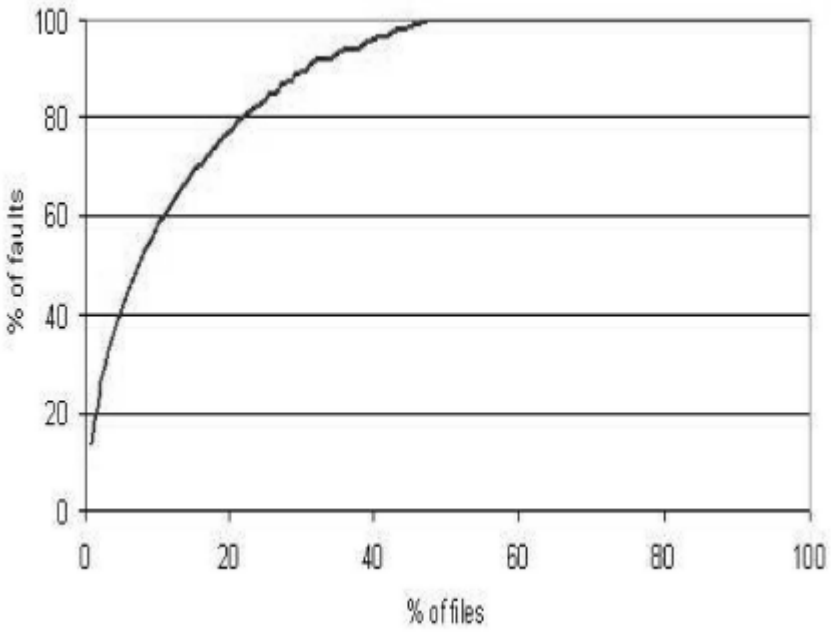

Figure 4. Pareto diagram for GCC [12]

The result of common resources of failures analysis on NASA mission can be seen in Figure 5. Requirements and coding faults are the most common fault types. Interestingly, design faults are less than 6% of all faults.

Source of Failures	% of SCRs
Requirements fault	32.65
Design fault	5.60
Coding fault	32.58
Data problem	13.72
Integration fault	2.27
I/O problem	0.63
Compilier/linker/sfw dev or test tool error	0.63
Simulation problem	0.21
Procedural non–compliance	1.54
Process problem	2.34
Fabrication/Manufacturing fault	0.45
None identified	7.38

Figure 5. Sources of failures for NASA mission [12]

In particular, it is claimed that requirements, coding and data faults are the three most common fault types. Rather than the common belief, most of the faults are related with the late phases of software development process.

3.5. Combinatorial Testing

Combinatorial testing is a very popular and an effective testing strategy. In this section we analyze the relation between faults and interaction strength, the AETG System and Combinatorial Interaction Testing.

The study [13] 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. The below items guide the research:

- "*Are the complex Mandelbugs of higher interaction strength than the deterministic Bohrbugs?*
- *And if so, how large is the difference?*
- *Does it take longer to find the more complex bugs?*
- *Implications for testers using combinatorial testing methods*" [13]

In this study, 242 bugs from bugs.mysql.com for MySQL software were examined. 35 of them could not be classified as Bohrbug or Mandelbug. 25% of all the bugs did not have enough information to specify the number of interaction to cause the fault. The results of the analysis of remaining data are shown in Figure 6. Mandelbugs have higher interaction strength than Bohrbugs. This means that more factors are needed to trigger more complex errors.

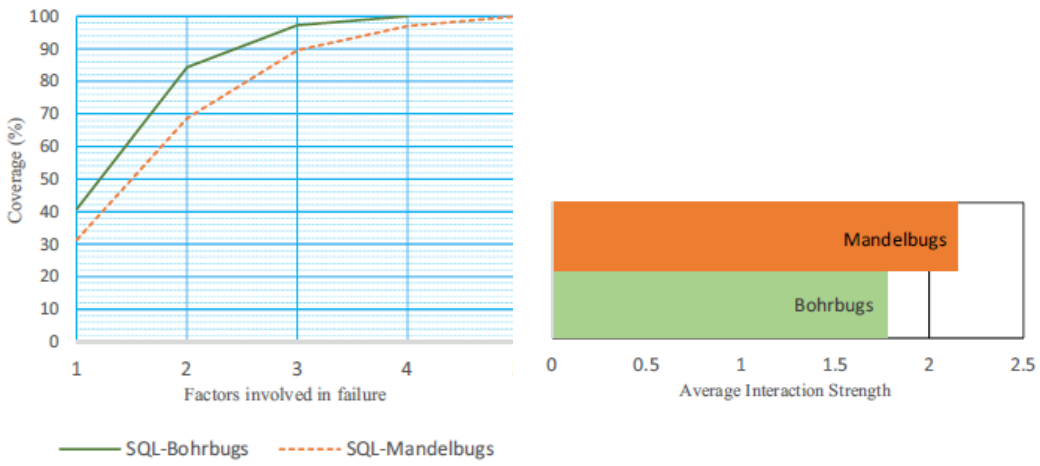

Figure 6. Interaction strength and Average interaction strength for MySQL bugs [13]

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 [14]. 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.

AETG was used to generate test cases for a system with 61 parameters, 29 of them with 2 values, 17 with 3 values and 15 with 4 values. Total combinations of these parameters result in 7.4×10^{25} test cases. AETG system produced 41 test cases for pairwise combinations. In the next release of the same system, there were 75 parameters, 35 of them with 2 values, 39 with 3 values and 1 with 4 four values. Exhaustive testing of all the combinations would bring about a total of 5.5×10^{29} test

cases. In this case, AETG generated 28 pairwise test cases. Although the second case had more parameters than the first one, having less 4 valued parameters helped the second case to produce less test cases in total. The logarithmic growth features of AETG algorithm can be clearly seen from this outcome.

In conclusion, 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 [15]. Four phases consists of Modeling, Sampling, Test and Analyze, which can be seen in Figure 7. Modeling and Sampling phases define "What" to test, and Test and Analyze phases define "How" to test.

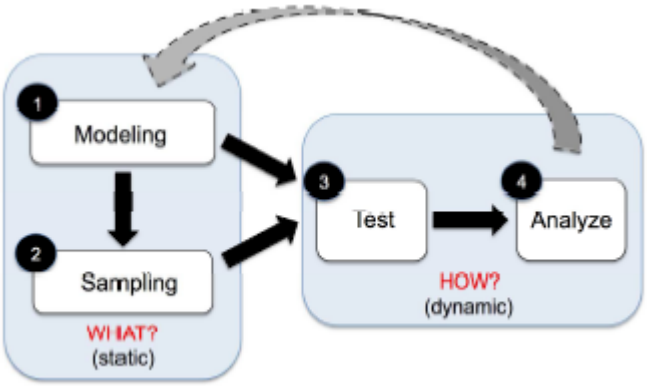

Figure 7. Four phases of CIT [15]

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.

Sampling is choosing the test data from the model defined in phase 1, with the help of an algorithm or a process. In this phase, test cases are generated according to a combinatorial approach and coverage criteria. To illustrate, selection of the interaction strength between system parameters, such as pairwise or 3-way etc., is done in Sampling phase. Existing sampling tools suffer from the lack of flexible and generic scenario generation. If there were a way to define an abstract model for users to build their own criteria, it would be more efficient and applicable.

In Testing, the inputs and the samples are utilized to run the tests. It can be done in one shot or incrementally. Throughout the process, there may be some issues with managing the schedule and test case prioritization. Even if the plan and the model are carefully designed, testers may face some unpredictable problems during test phase. To overcome problems like these, test approach should be incremental and adaptive.
After test phase, Analysis is done on test results to see if tests are failed or passed. Fault localization and fault characterization are performed on the parameters and values that caused the failure. To improve this process, it is important to support the analysis with tool that can combine different fault characterization approaches.

CHAPTER 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. Detailed project size metrics according to domain and software type are given in Table 5.

Table 5. Average size metrics per CSCI according to domain and software type

Domain	Software type	Requirements	Lines of Code
Electronic Warfare	Embedded	500	63500
	GUI	575	135000
Radar	Embedded	150	30000
	GUI	375	57000

In total, there were 1461 faults. We collected 10 features for each fault, shown in Table 6.

Table 6. Fields of faults and their possible values

Field	Possible values
Detailed explanation of the fault	Free text
Detection phase	CSCI test, Preparation test, Acceptance test, System integration test, System test
Fault type	Functional, Performance, Interface, Other
Severity	1-3 (1 is most critical)
Reproducibility	Seen only once, Generated randomly, Regularly reproducible, Unknown
Source of the fault	System requirements, System design, Software requirements, Software design, Coding, Environment, Test
Solution type	Code correction, Document update, Test environment correction
Effort for solution	In minutes
Verification method	Analysis, Test, Demonstration, Document check
Effort for verification	In minutes

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 answers to certain questions. Some of these questions are gathered from the trending topics in the literature and some of them are triggered by the company developing these software projects. The questions are:

- What is the distribution of the faults according to detection phase?
- What is the distribution of the faults according to fault type?
- What is the distribution of the faults according to source of the fault? [12] [16]
- What is the effort for solution according to source of the fault?
- What is the effort for solution of the most critical faults according to detection phase? [17]
- What is the number of factors involved in failures? [13]

4.1. Distribution of the faults according to detection phase

In the development life cycle, software start being tested at CSCI level. At this phase, one CSCI is tested alone while surroundings are just mock software that behave similar to real ones. After CSCI tests, two or more CSCIs that communicate with each other are brought together in System Integration Test phase to construct smaller system parts and find the bugs that may be triggered by the communication of different CSCIs. When these two steps are done, the system is fully operated at System test phase to analyze whether it works in accordance with the system requirements. Before Acceptance Test phase with the customer, a Preparation test is done as a walk through over the test procedure. Finally, acceptance test is done with the customer.

We wanted to do an analysis to see at what point of the project, most of the faults were found. Common approach suggests that detection rate of bugs ideally drops towards the end of the software development life cycle. However, it is not always the case. In our analysis, which can be seen in Figure 8, when integrating different CSCIs, 38,06% of all faults were triggered. As the number of CSCIs increased, the communication overhead and the bugs due to issues of connecting different CSCIs also increased. The faults found at Acceptance test seemed to be quite high. This usually happens when customer does not get involved in the development of the software at the early phases, which leads to misunderstandings and a high fault rate at the acceptance phase. Looking at these results, improving CSCI test performance may help reduce the risks with less effort.

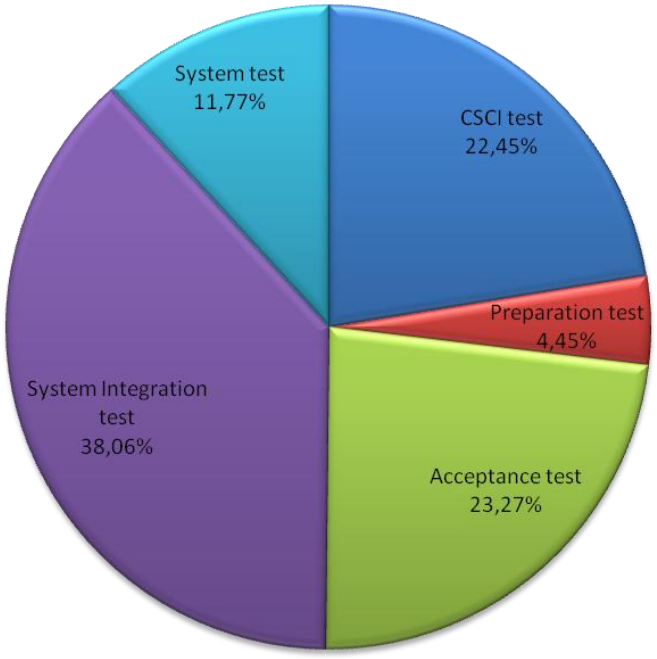

Figure 8. Distribution of the faults according to detection phase

4.2. Distribution of the faults according to fault type

Types of faults may have an effect on the solution type and effort. If one of the types is more common than others, then applying special techniques for finding that type of faults can reduce costs in total.

When we analyzed the distribution of fault types which can be seen in Figure 9, 87,41% of all faults were functional. It surely infer that finding functional faults at CSCI test should be the first priority in order to increase effectiveness.

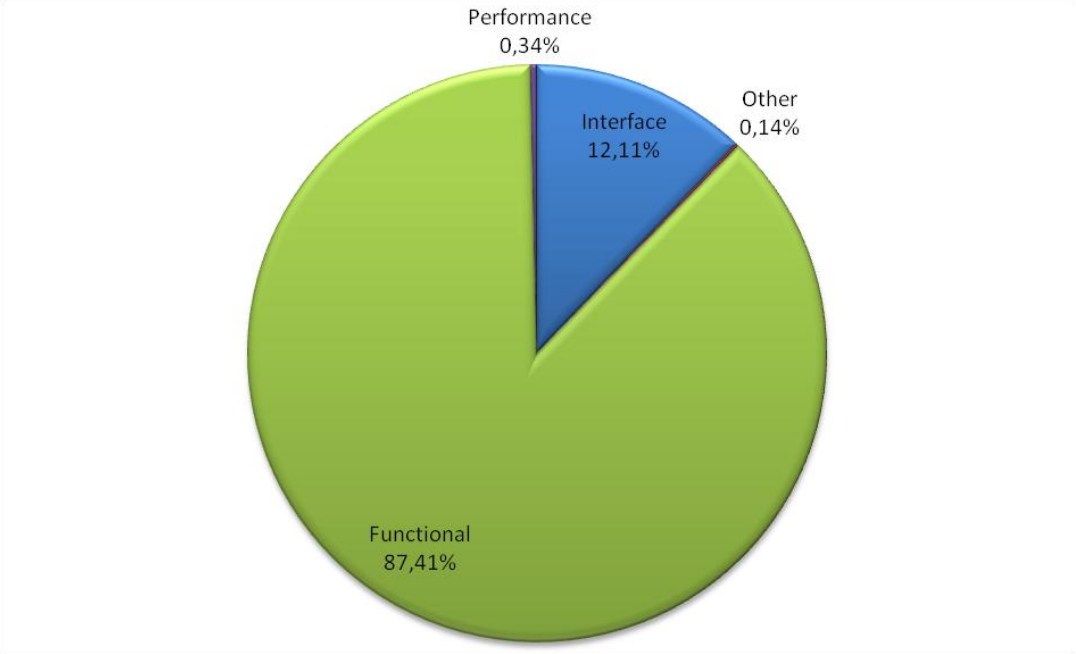

Figure 9. Distribution of the faults according to fault type

4.3. Distribution of the faults according to source of the fault

In [12], the question "Are some sources of failures more common than others?" was asked while analyzing fault data. Their results have shown that the most common sources were requirements faults with 32,65% and coding faults with 32,58% of all faults. Then data problems followed them with 13,72%.

In another paper [16], the fault data was analyzed in the form of "Defect Origins vs. Defects per Function Point". Coding faults were first with 1,75 defects per function point, then design faults and requirement faults followed it with 1,25 and 1 defects per function point respectively.

In this perspective, we applied this analysis to see the sources that caused the faults in detail. In our data, each fault had only one corresponding source which has the primary responsibility for triggering it. Coding was the first source that is responsible for causing a fault with a huge difference than others. The other sources and the distribution can be seen in Figure 10.

According to these results, coding activities should be focused and analyzed in detail.

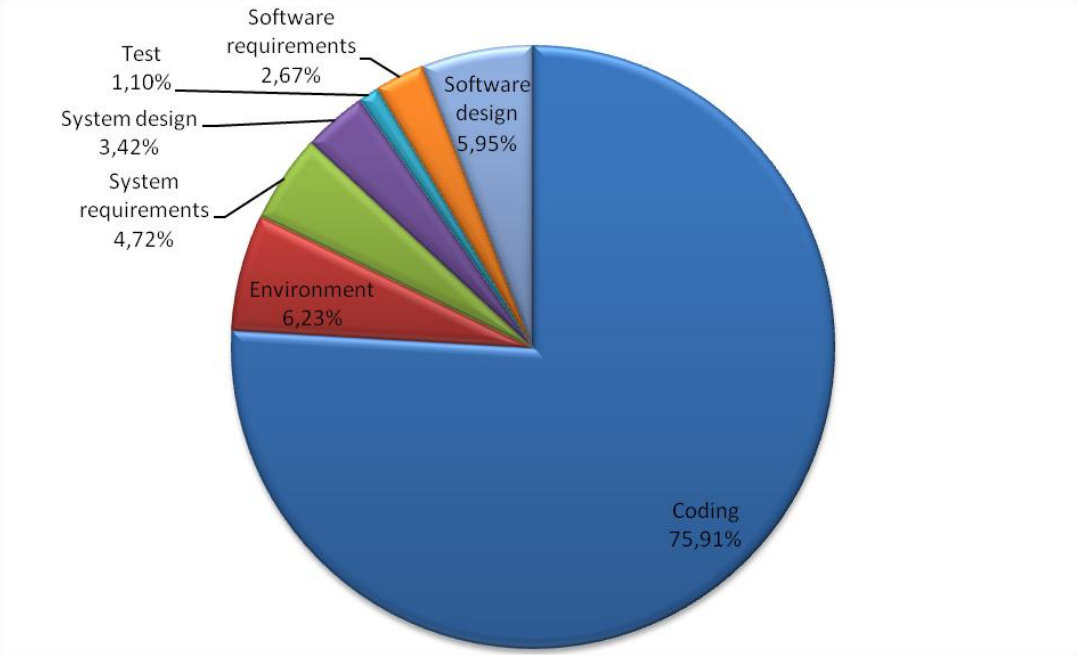

Figure 10. Distribution of the faults according to source of the fault

4.4. The average effort for solution according to source of the fault

The source of the fault itself could not be enough to explain everything. Therefore, we moved on analyzing the effect of the sources on effort for solution. The average effort for solution vs. sources of the fault graph can be seen in Figure 11. Design problems clearly required more effort than other sources. Then, requirements and coding followed design problems. It seemed that errors in early phase activities resulted in more effort for solving them. It justifies the idea that risks should be handled in early phases to reduce costs.

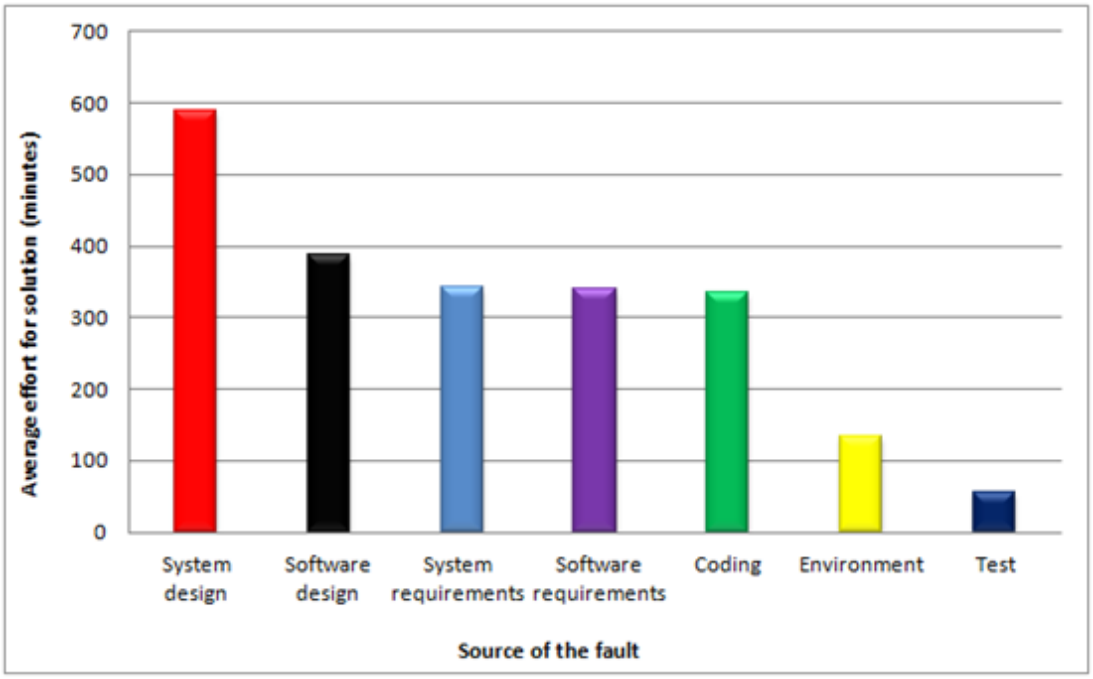

Figure 11. The average effort for solution according to source of the fault

4.5. The average effort for solution of the most critical faults according to detection phase

Another important issue was the incidence of critical faults at later stages in the project. It could cost more as the project proceeded towards the end. Software defect reduction top-10 list in [17] suggests that "Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase." It makes an emphasis on the importance of early phases like requirement analysis and design, early verification and validation to reduce the effort and cost of bug fixes. They use the "often" word to point out some exceptional cases. In small and noncritical software systems, this ratio is 5:1. Even large and critical software can have a smaller ratio with the help of good architectural practices. An example project that has well encapsulated modules in [18] has a ratio of 2:1.

These ideas inspired us to do an analysis of the average effort for solution of the most critical faults according to detection phase. Looking at the results seen in Figure 12, the average effort for solving the most critical faults at early phases seemed to be easier. In CSCI test, it was way cheaper to handle critical faults than any other phase.

Preparation test had the highest average effort for solving a critical fault, which was about 3 times higher than CSCI test phase.

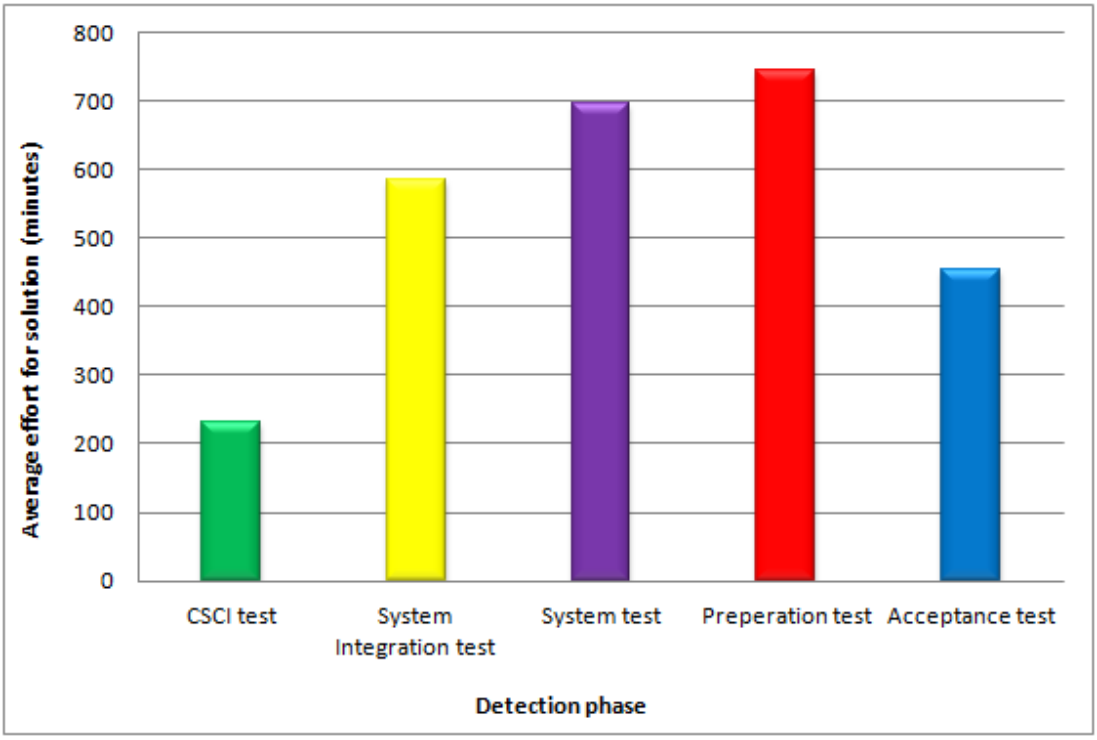

Figure 12. The average effort for solution of the most critical faults according to detection phase

4.6. The number of factors involved in failures

An analysis of the relation between t-way interaction of input parameters and the bug types are made in [13]. The average and the total interaction strength for all bugs are collected. The results of the analysis show that the previous inferences about 6-way combinatorial testing being effective as exhaustive testing also apply to their case.

We investigated the number of factors involved in failures in our fault data as well. 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. In Figure 13, 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.

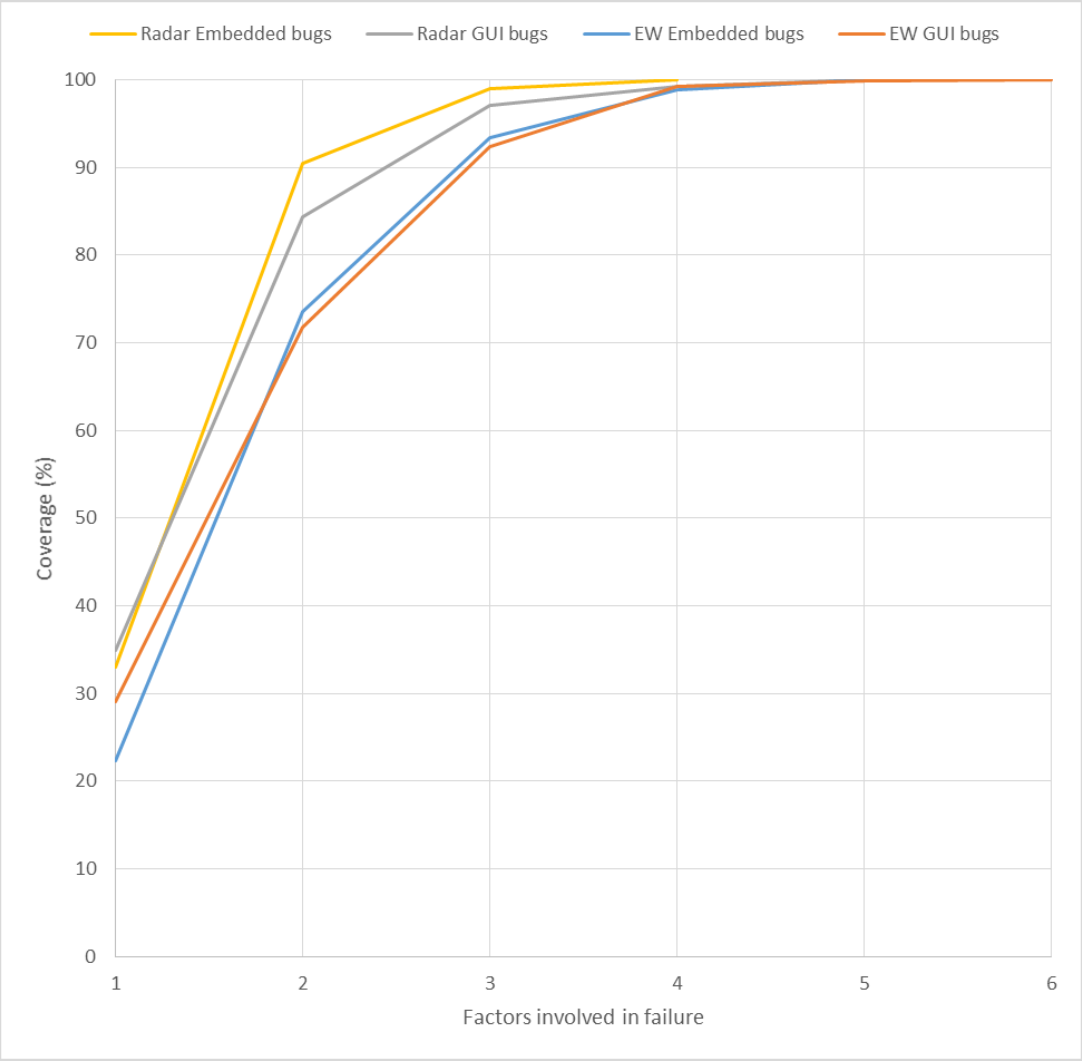

Figure 13. Interaction strength for EW and Radar bugs

Average interaction strength for all bug types can be seen in Figure 14. EW bugs seem to have a higher average interaction strength than Radar bugs, indicating that it

is more difficult to trigger EW bugs than Radar bugs. In EW domain, the average of factors involved in failure for Embedded bugs is higher than GUI bugs, whereas in Radar domain, GUI bugs require more effort to be triggered than Embedded bugs.

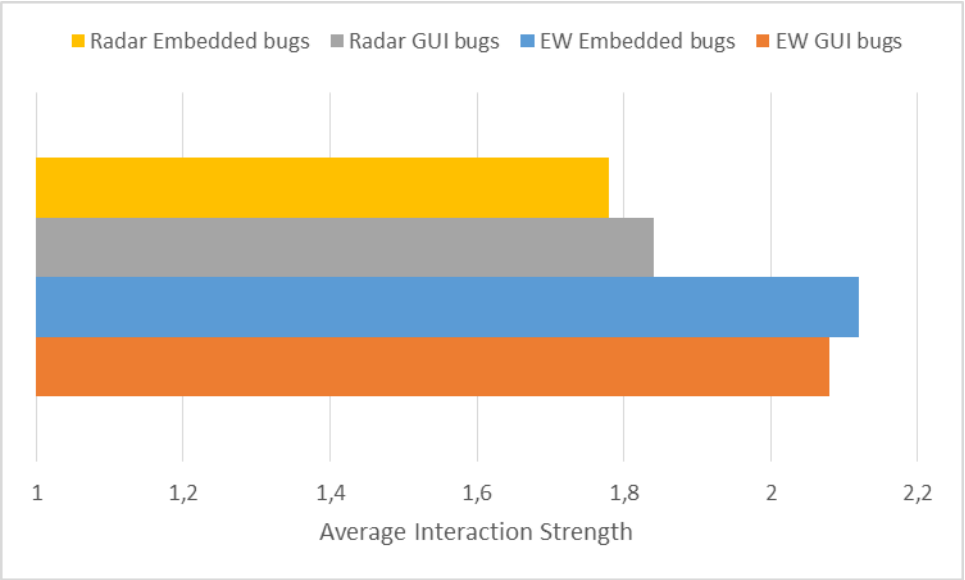

Figure 14. Average interaction strength for EW and Radar bugs

CHAPTER 5

COMBINATORIAL TEST DATA GENERATOR

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 made in [13], we decided to use combinatorial testing approach which is known to be an effective testing strategy [1] [14]. Because the analysis yielded the importance of functional tests, we needed to find an effective approach for functional tests as well.

First we investigated the combinatorial test data generator tools available. Most of the existing tools were built for pairwise testing [20] (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 [19] seemed to be suitable for these constraints and performed better than its counterparts [20]. 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.

ACTS uses an algorithm called In-Parameter-Order-General (IPOG), which is the generalized form of In-Parameter-Order algorithm from pairwise to t-way testing. The details of IPOG algorithm can be seen in Figure 15. For a system with t or more parameters, IPOG generates all combinations of first t parameters (lines 1-3), then extends this set by one parameter until a t-way set is generated for all parameters (lines 4-19). The extension part is done in two steps; horizontal (lines 6-10) and vertical growth (lines 11-18).

```
Algorithm IPOG-Test (int t, ParameterSet ps)
{
1. initialize test set ts to be an empty set
2. denote the parameters in ps, in an arbitrary order, as P1, P2, ..., and Pn
3. add into test set ts a test for each combination of values of the first t parameters
4. for (int i = t + 1; i ≤ n; i ++){
5.    let π be the set of t-way combinations of values involving parameter Pi
        and t -1 parameters among the first i – 1 parameters
6.    // horizontal extension for parameter Pi
7.    for (each test τ = (v1, v2, ..., vi-1) in test set ts) {
8.       choose a value vi of Pi and replace τ with τ' = (v1, v2, ..., vi-1, vi) so that τ' covers the
           most number of combinations of values in π
9.       remove from π the combinations of values covered by τ'
10.   }
11.   // vertical extension for parameter Pi
12.   for (each combination σ in set π){
13.     if (there exists a test that already covers σ) {
14.        remove σ from π
15.     } else {
16.        change an existing test, if possible, or otherwise add a new test
             to cover σ and remove it from π
17.     }
18.   }
19.}
20.return ts;
}
```

Figure 15. Algorithm IPOG-Test [19]

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. Class diagram for the tool can be seen in Figure 16.

Our design structure consists of 3 stages: reading the input from a file, discretizing the parameter values and generating the combinatorial test data using ACTS tool.

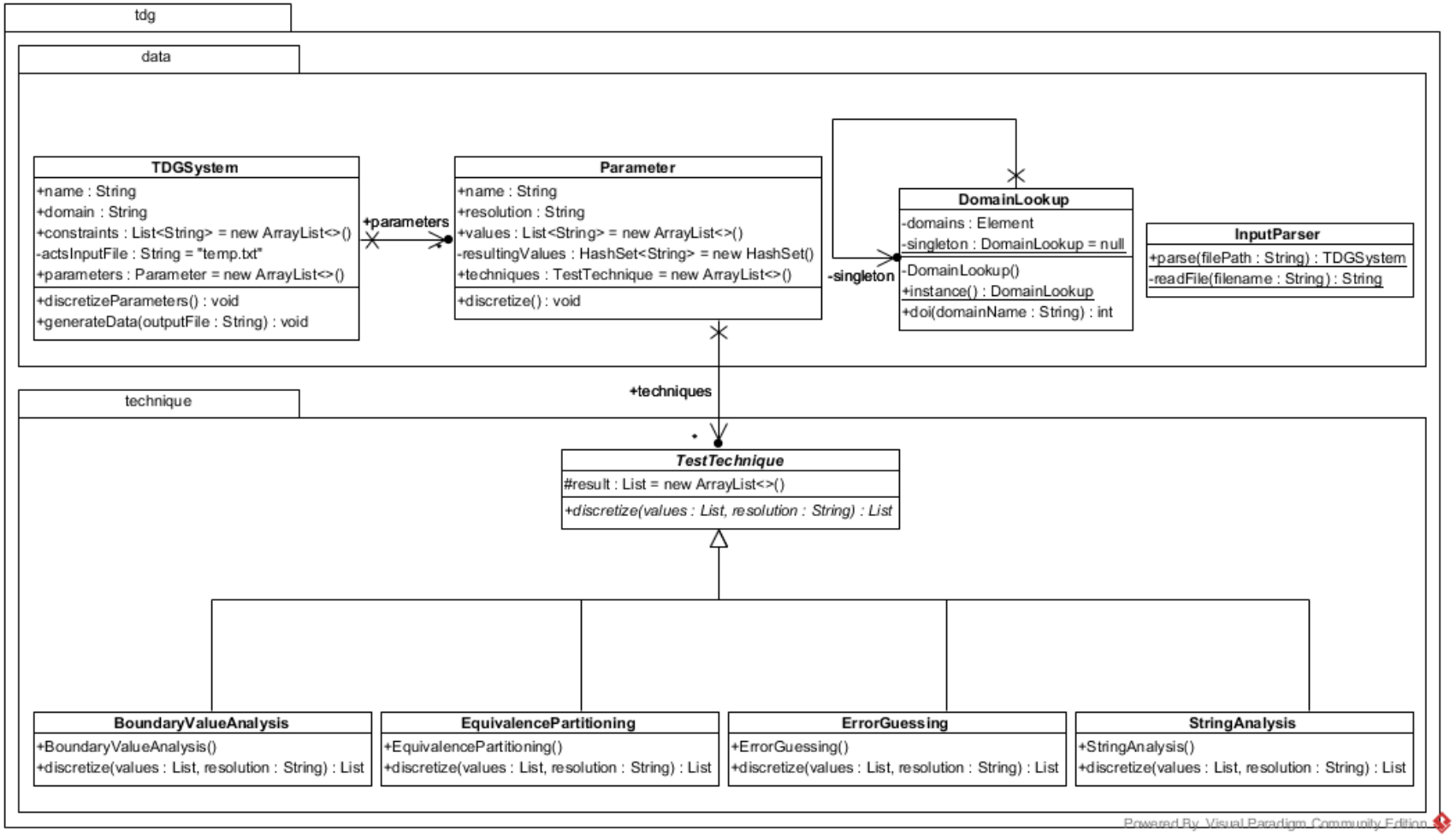

Figure 16. Class Diagram for Test Data Generator tool

5.1.1. 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 3 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 become 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, generated degree of interaction “doi” for each domain can be seen in Figure 17. 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.

```
<?xml version="1.0" encoding="UTF-8"?>
<Domains>
    <EW_GUI doi="3"></EW_GUI>
    <EW_Embedded doi="3"></EW_Embedded>
    <Radar_GUI doi="3"></Radar_GUI>
    <Radar_Embedded doi="2"></Radar_Embedded>
</Domains>
```

Figure 17. Content of domain.xml file

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. Content of a json input file and an xml input file can be seen in Figure 18 and Figure 19 respectively.

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. The constraints section allows users to specify conditions that will be taken into account during test data generation.

Figure 18. Content of a json input file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<System name="XXXX" domain="EW_GUI">
  <Parameters>
    <Parameter name="p0" resolution="0.1">
      <values>
        <value>0</value>
        <value>550</value>
        <value>1000</value>
      </values>
      <techniques>
        <technique>ErrorGuessing</technique>
        <technique>EquivalencePartitioning</technique>
        <technique>BoundaryValueAnalysis</technique>
      </techniques>
    </Parameter>
    <Parameter name="p1">
      <values>
        <value>2</value>
        <value>3</value>
        <value>4</value>
        <value>5</value>
      </values>
      <techniques>
        <technique>ErrorGuessing</technique>
      </techniques>
    </Parameter>
    <Parameter name="p2" resolution="7">
      <values>
        <value>asdf</value>
        <value>qwerty</value>
      </values>
      <techniques>
        <technique>ErrorGuessing</technique>
        <technique>StringAnalysis</technique>
      </techniques>
    </Parameter>
  </Parameters>
  <Constraints>
    <Constraint text="p0>p1 =&gt; p2!=&quot;asdf&quot;">
      <Parameters>
        <Parameter name="p0" />
        <Parameter name="p1" />
      </Parameters>
    </Constraint>
  </Constraints>
</System>
```

Figure 19. Content of an xml input file

Our tool parses these input files using built-in Java xml libraries and an open source json library [21]. Then a TDGSystem object and its related Parameter and TestTechnique objects are created using InputParser class.

5.1.2. 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. All of these techniques extend the abstract *TestTechnique* class (see Figure 16) and each of them implements the *discretize(List values, String resolution)* method according to their way of making parameter values discrete. Due to this structure, these techniques can be extended easily just by adding a new class that extends *TestTechnique* and implementing the *discretize(List values, String resolution)* method. Then, in the input file, the name of this class should be given in techniques of the parameters section.

EquivalencePartitioning class implements the *discretize(List values, String resolution)* method by selecting a random value inside the given range of the parameter taking resolution into consideration. *BoundaryValueAnalysis* class selects min and max values in the range of the parameter, also (max+resolution) and (min-resolution) values. *ErrorGuessing* class just selects the values given in the input file. This class was added to give the user the ability to add values that must be tested according to their experience. *StringAnalysis* class 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, *discretizeParameters()* method of *TDGSystem* class is called. The sequence diagram for this method can be seen in Figure 20. In *discretizeParameters()* method, *discretize()* method for each parameter in the system is called. Then for every technique of the parameter, *discretize(List values, String resolution)* method is invoked. This method returns the list of discrete values that the technique produces. When the loop finishes, the field *values* of Parameter object is assigned with the new discrete values. After running the same procedure on all the parameters in the system, all continuous and undetermined values are transformed into specific, discrete and meaningful data.

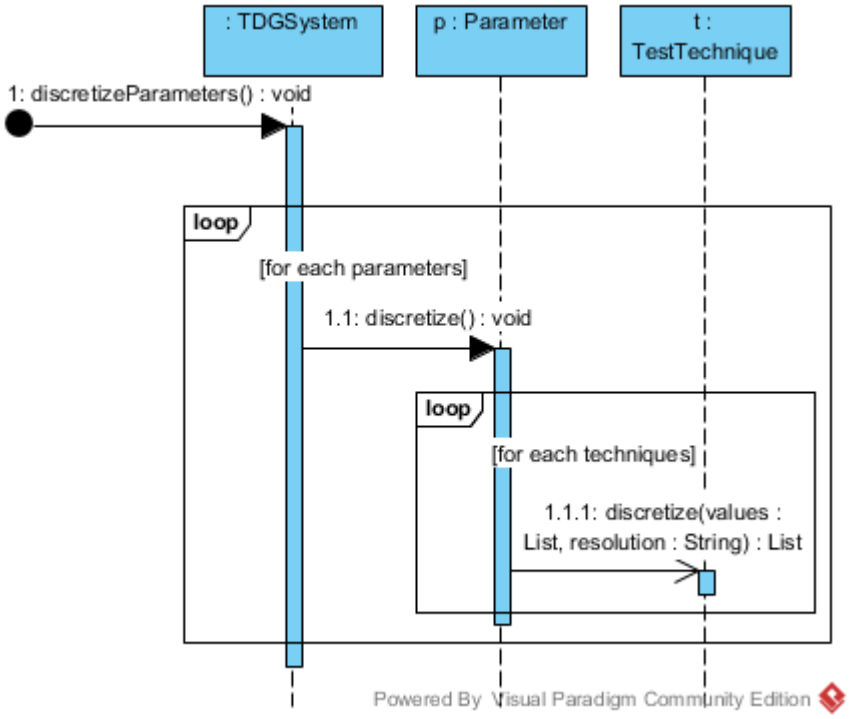

Figure 20. Sequence diagram for discretizing the parameter values

5.1.3. Generating the combinatorial test data

Once we obtained 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. An example of an ACTS input file can be seen in Figure 21. It contains the system name, parameter names, types and their range, and if exists, the constraints information.

```
[System]
Name: C1

[Parameter]
p1(int): 0,1,2,3
p2(int): 0,1,2,3
p3(int): 0,1,2,3
p4(enum): a, s, d, 34, 56
p5(boolean): TRUE, FALSE

[Constraint]
p1>p2 || p3>p2
```

Figure 21. Content of an ACTS input file

For the purpose of using ACTS tool, we generate a temporary input file with the information we have in *TDGSystem* class (see Figure 16). When *generateData(String outputFile)* method of the object of class *TDGSystem* is called, first the system name, parameter values and constraints are written in a file, as seen in the sequence diagram of generating the combinatorial test data in Figure 22.

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). The *doi(String domainName)* method of the class *DomainLookup* parses the domain.xml file mentioned in Section 5.1.1 and returns 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. Inside our tool, we call *Runtime.exec()* method and give required command line parameters to benefit from this feature of ACTS. We give the generated temporary input file path "actsInputFile", degree of interaction value "doi" gathered for the SUT, and the output file path "outputFile" where the generated test data will be written (see Figure 22). ACTS supports generating the output in four different formats: numeric, nist,

csv, excel. This option can also be configured easily by giving the command line parameter “Doutput”. In our case, the output file is generated in “CSV” format. With this final step, generation of t-way combinatorial test data for our system is completed.

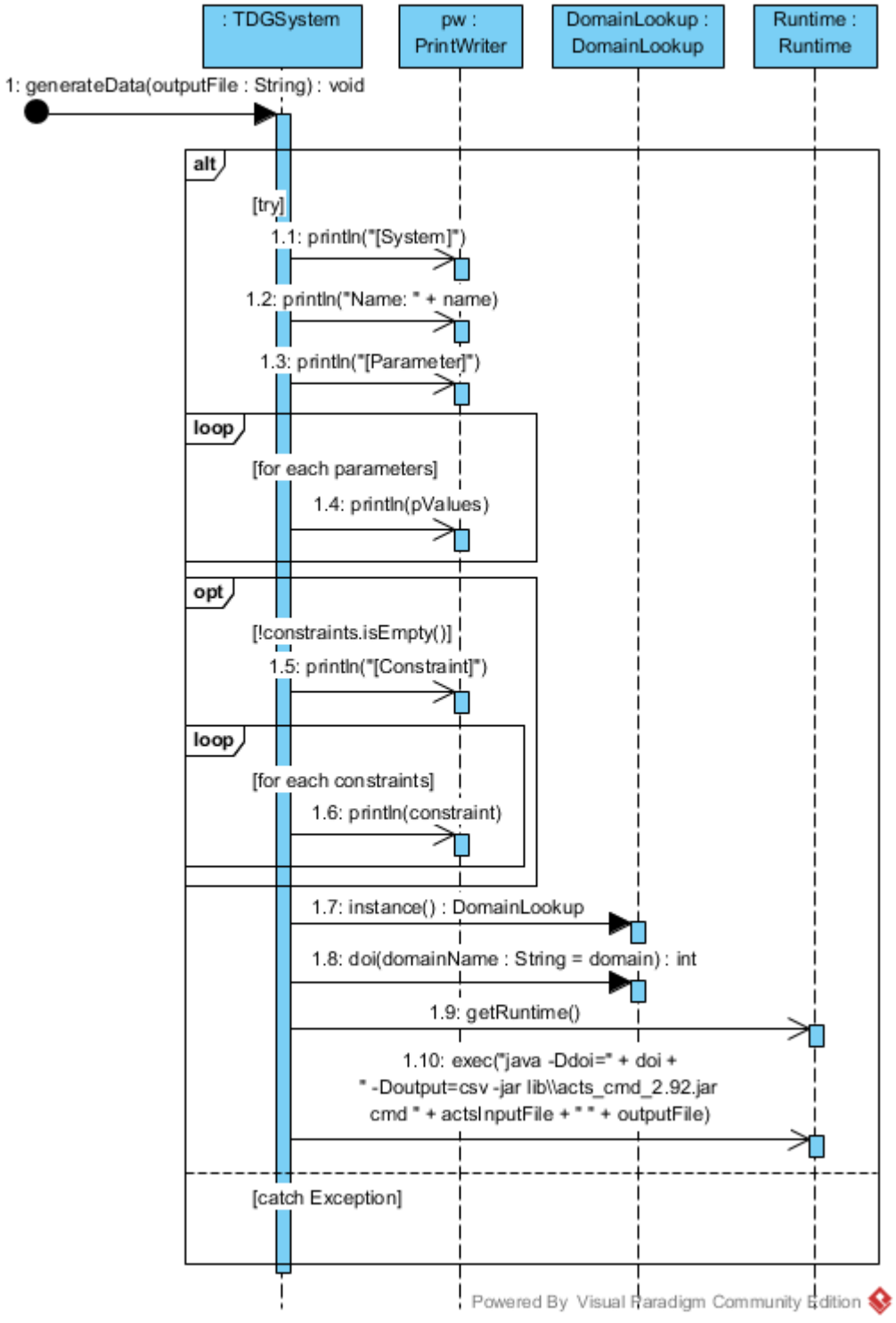

Figure 22. Sequence diagram for generating the combinatorial test data

CHAPTER 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 Figure 23, became fully automatic.

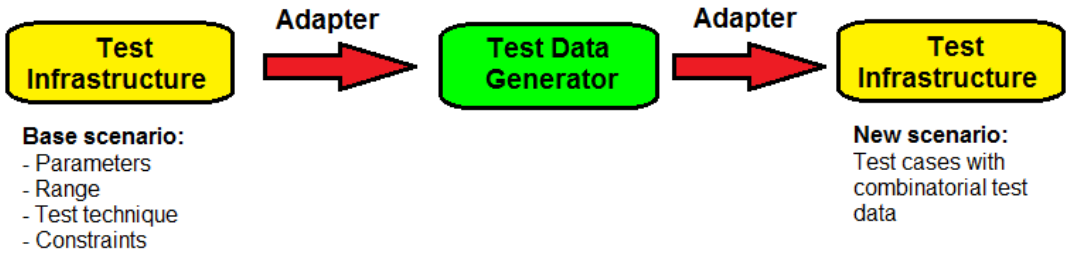

Figure 23. Test data generation cycle

The adapter software converts the parameter information inside the base scenario into an input file for the test data generator tool, mentioned in Section 5.1.1. 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 mentioned in Section 5.1.1. First, the tool discretizes the possible values of the parameters according to test techniques and then generates the combinatorial test data according to constraints, mentioned in Section 5.1.2 and 5.1.3. 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 research questions were:

- RQ1: How long did it take to prepare input files?

- RQ2: How long did it take to carry out tests?
- RQ3: How many tests were generated?
- RQ4: How many bugs were triggered?
- RQ5: How many new bugs were found?
- RQ6: 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 detailed results and comparisons can be seen in Figure 24, 25 and 26. 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%.

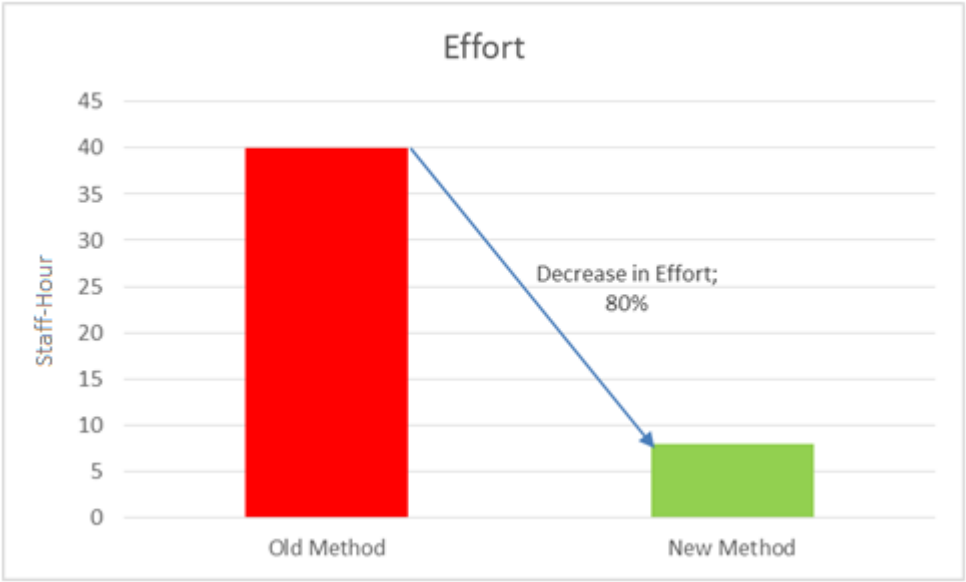

Figure 24. Comparison of effort in staff-hour

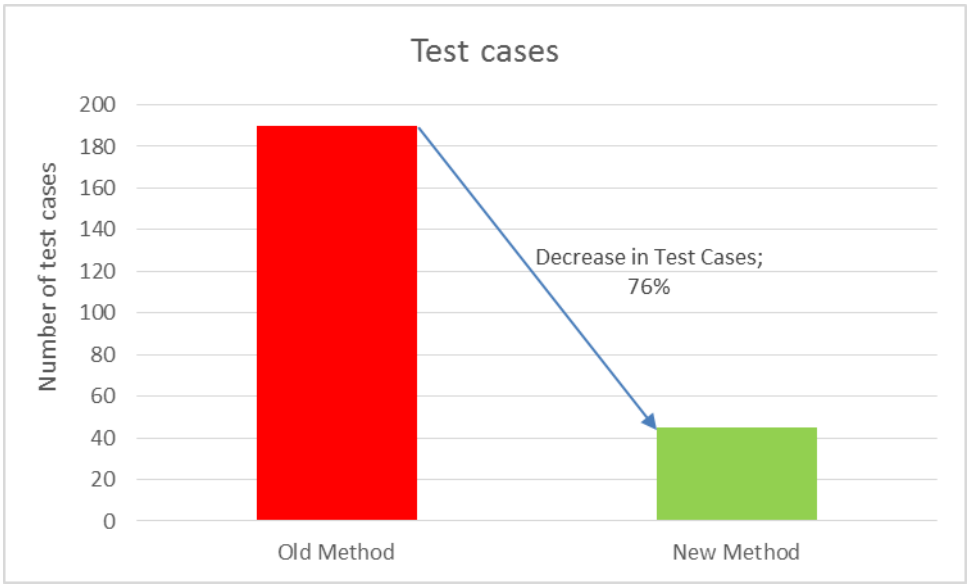

Figure 25. Comparison of number of test cases

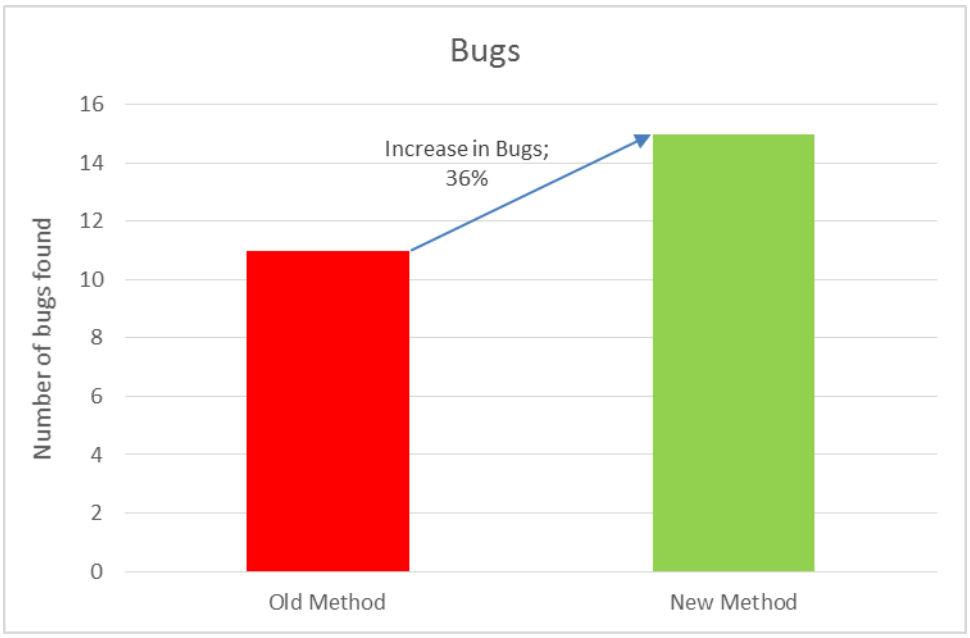

Figure 26. Comparison of number of bugs found between old method and new method

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. The comparison of two cases can be seen in Figure 27.

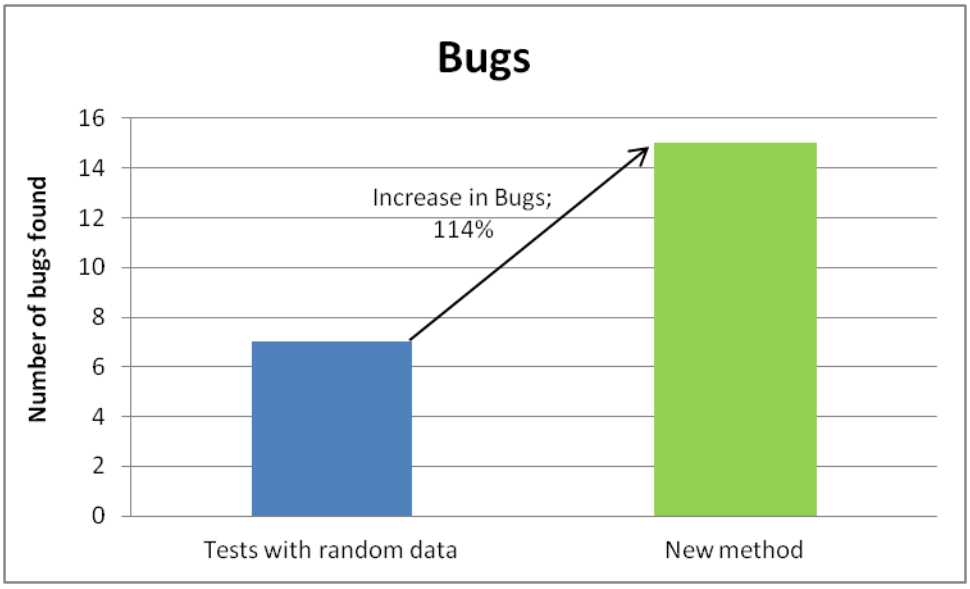

Figure 27. Comparison of number of bugs found between tests with random data and new method

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 5, 6 and 7 are explained in detail.

CHAPTER 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

[1] Kuhn, R., Kacker, R., Lei, Y., & Hunter, J. (2009). Combinatorial Software Testing. Computer, 42(7), 94-96.

[2] Mariani, L. (2003). A Fault Taxonomy for Component-Based Software. Electronic Notes in Theoretical Computer Science, 82(6), 55-65.

[3] Bruning, S., Weissleder, S., & Malek, M. (2007). A Fault Taxonomy for Service-Oriented Architecture. 10th IEEE High Assurance Systems Engineering Symposium (HASE07).

[4] Chan, K. S., Bishop, J., Steyn, J., Baresi, L., & Guinea, S. (2009). A Fault Taxonomy for Web Service Composition. Service-Oriented Computing - ICSOC 2007 Workshops Lecture Notes in Computer Science, 363-375.

[5] V. H. Nasser, W. Du, and D. MacIsaac, "An ontology-based software test generation framework," in Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering, 2010, pp. 192-197.

[6] J. Ploski, M. Rohr, P. Schwenkenberg, and W. Hasselbring. Research issues in software fault categorization. SIGSOFT Softw. Eng. Notes, 32(6):6, 2007.

[7] Donald E. Knuth. The errors of TEX. Software Practice and Experience, 19(7):607–685, July1989.

[8] Boris Beizer and Otto Vinter. Bug taxonomy and statistics. Technical report, Software Engineering Mentor, 2630 Taastrup, 2001.

[9] Jim Gray. Why do computers stop and what can be done about it? In Proceedings of Symposium on Reliability in Distributed Software and Database Systems (SRDS-5), pages 3–12. IEEE CS Press, 1986.

[10] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M. Y. Wong. Orthogonal defect classification-a concept for in-process measurements. IEEE Transactions on Software Engineering, 18(11):943–956, 1992.

[11] Richard A. DeMillo and Aditya P. Mathur. A grammar based fault classification scheme and its application to the classification of the errors of TEX. Technical report, Software Engineering Research Center and Department of Computer Sciences, Purdue University, 1995.

[12] M. Hamill, K. Goseva-Popstojanova, "Common Trends in Software Fault and Failure Data", IEEE Trans. Software Eng., vol. 35, no. 4, pp. 484-496, July/Aug. 2009.

[13] Z. Ratliff, R. Kuhn, R. Kacker, Y. Lei, K. Trivedi, "The Relationship Between Software Bug Type and Number of Factors Involved in Failures", Intl Wkshp Combinatorial Testing, 2016.

[14] D.M. Cohen, S.R. Dalal, M.L. Fredman, G.C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design", IEEE Trans. Software Eng., vol. 23, no. 7, pp. 437-444, July 1997.

[15] C. Yilmaz, S. Fouch´e, M. B. Cohen, A. A. Porter, G. Demiröz, and U. Koc. Moving forward with combinatorial interaction testing. IEEE Computer, 47(2):37–45, 2014.

[16] Fenton, N., & Neil, M. (1999). A critique of software defect prediction models. IEEE Transactions on Software Engineering, 25(5), 675-689.

[17] Basili, V. R., & Boehm, B. W. (2009). Foundations of empirical software engineering: the legacy of Victor R. Basili; with 21 tables. Berlin: Springer.

[18] Royce, W. (2003). Software project management: a unified framework. Reading: Addison-Wesley.

[19] Lei, Y., Kacker, R., Kuhn, D. R., Okun, V., & Lawrence, J. (2007). IPOG: A General Strategy for T-Way Software Testing. 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS07).

[20] Pairwise Testing Available Tools. Retrieved February, 2018, from http://www.pairwise.org/tools.asp.

[21] JSON. Retrieved March, 2018, from http://www.json.org.

TEZ FOTOKOPİ İZİN FORMU

ENSTİTÜ

Fen Bilimleri Enstitüsü ☐

Sosyal Bilimler Enstitüsü ☐

Uygulamalı Matematik Enstitüsü ☐

Enformatik Enstitüsü ■

Deniz Bilimleri Enstitüsü ☐

YAZARIN

Soyadı : Bosnalı
Adı : Hakan
Bölümü : Bilişim Sistemleri

TEZİN ADI (İngilizce) : A COMBINATORIAL TEST DATA GENERATION APPROACH USING FAULT DATA ANALYSIS AND DISCRETIZATION OF PARAMETER INPUT SPACE

TEZİN TÜRÜ: Yüksek Lisans ■ Doktora ☐

1. Tezimin tamamı dünya çapında erişime açılsın ve kaynak gösterilmek şartıyla tezimin bir kısmı veya tamamının fotokopisi alınsın. ■
2. Tezimin tamamı yalnızca Orta Doğu Teknik Üniversitesi kullanıcılarının erişimine açılsın. (Bu seçenekle tezinizin fotokopisi ya da elektronik kopyası Kütüphane aracılığı ile ODTÜ dışına dağıtılmayacaktır.) ☐
3. Tezim bir (1) yıl süreyle erişime kapalı olsun. (Bu seçenekle tezinizin fotokopisi ya da elektronik kopyası Kütüphane aracılığı ile ODTÜ dışına dağıtılmayacaktır.) ☐

Yazarın imzası: Tarih: 22.06.2018