Use of Orthogonal Arrays and Design of Experiments via Taguchi methods in Software Testing

LJUBOMIR LAZIĆ
State University of Novi Pazar
Vuka Karadžića bb, 36 300 Novi Pazar, SERBIA
llazic@np.ac.rs , http://www.np.ac.rs

Abstract: - To solve the problem of great number of test cases, and to force the configuration testing to be effective, combinatorial testing is proposed, using an Orthogonal Array Testing Strategy (OATS) as a systematic, statistical way of testing pair-wise interactions. This combinatorial approach to software testing uses models to generate a minimal number of test inputs so that selected combinations of input values are covered. The OAT method can simultaneously reduce testing costs, product introduction delays, and faults going to the field by generating test cases that are more efficient and thorough in finding faults. Often the result is a 50% reduction in the number of tests and detection of more faults. An advantage of the Taguchi method application in Software Testing is that it emphasizes a mean performance characteristic (Defect fixing time and cost of software Quality) value close to the target value rather than a value within certain specification limits, thus improving the product quality. Additionally, Taguchi's method for experimental design is straightforward and easy to apply as we did for defect Cost to fix [$] and Total Resolution time [Days] minimisation versus controlled factors: Severity, Complexity and engineers Experience to many engineering situations, making it a powerful yet simple tool.

Key-Words: - Software testing, Bug fixing, Resources allocation, Orthogonal Array, DOE, Taguchi method

1 Introduction

Our research [1][1] concluded that software development project employs some Quality Control (QC) process to detect and remove defects. The final quality of the delivered software depends on the effort spent on all the QC stages. Given a quality goal, different combinations of efforts for the different QC stages may lead to the same goal. For the quality of the final software we use the commonly used measure of delivered defect density - the number of defects present in the final product normalized by the size of the product. One of the main objectives of a project is to achieve the desired quality goal with least amount of resources. Using defects as the defining metric for quality, we can view the process of a project as comprising of defect injection and removal stages. There are some stages like the requirements, design and coding, in which defects are injected. These defects are removed in various QC stages. A QC stage can be characterized by the defect removal rate of that stage. There can be many possible combinations of defect removal rates for the different QC stages that can achieve the same overall quality goal. The different combinations will have different implications on the total QC effort. Clearly, for a process designer or a project manager, a key problem is to select the amount of effort to be spent in each QC stage such that the desired quality goal is met with the minimum cost. We propose a model i.e. OptimalSQM for the cost of QC process and then view the resource allocation among different QC stages as an optimization problem. Software testing consumes 30-70% of the development resources; however, shipped products may still have many residual faults resulting in low reliability, high usage cost, and high maintenance cost. For software testing process optimization we apply Orthogonal Array-Based Testing Strategy (OATS) and Design of Experiments via Taguchi method.

Different types of testing aims for identifying different types of errors and faults. For example, mutation testing modifies the source code in a meager way that helps the tester to develop effective test cases. Similarly, combinatorial testing is

[1] This work was supported in part by the Ministry of Education and Science of the Republic of Serbia under Grant No. TR-35026 entitled as:"Software Development Environment for optimal software quality design".

focused on identifying errors and faults, occurs due to the interaction of different parameters of software. Combinatorial testing is performed by covering all the possible combination of parameter values. Testing of a network gaming application running on the internet can be influenced by the number of parameters. These parameters are operating system, audio, graphics, number of players, internet access type, browser type, etc. Each parameter may have any number of possible values. The interaction of these parameters causes faults and errors in the application. Exhaustive testing is virtually impractical due to several possible combinations of parameters. Combinatorial Testing provides a better way to cover all the possible combinations with a better tradeoff between cost and time.

Combinatorial testing is based on the following concepts:

Interaction Rule: Most failures occur due to a single factor or by the joint combinatorial effect (interaction) of two factors, with progressively fewer failures induced by interactions between three or more factors [2].

T-way Testing/Pairwise Testing: Pair-wise testing [3] requires a given numbers of input parameters to the system, each possible combination of values for any pair of parameters covered with at least one test case.

Covering Array: Covering array represents the test case selected under pairwise testing [5].

Combinatorial testing is a vital approach to detect interaction errors occurs because of interaction of several parameters. There are two approaches for combinatorial testing:

- Testing of configuration parameter values, or
- Testing of input parameter values.

To solve the problem of great number of test cases, and to force the configuration testing to be effective, combinatorial testing is proposed, using an OAT Strategy as a systematic, statistical way of testing pair-wise interactions. This combinatorial approach to software testing uses models to generate a minimal number of test inputs so that selected combinations of input values are covered. The OAT method can simultaneously reduce testing costs, product introduction delays, and faults going to the field by generating test cases that are more efficient and thorough in finding faults. Often the result is a 50% reduction in the number of tests and detection of more faults.

An advantage of the Taguchi method application in Software Testing is that it emphasizes a mean performance characteristic (Defect fixing time and cost of software Quality) value close to the target value rather than a value within certain specification limits, thus improving the product quality. Additionally, Taguchi's method for experimental design is straightforward and easy to apply to many engineering situations, making it a powerful yet simple tool. It can be used to quickly narrow down the scope of a research project or to identify problems in a manufacturing process from data already in existence. Also, the Taguchi method allows for the analysis of many different parameters without a prohibitively high amount of experimentation. For example, a process with 8 variables, each with 3 states, would require 6561 (3^8) experiments to test all variables. However using Taguchi's orthogonal arrays, only 18 experiments are necessary, or less than 0.3% of the original number of experiments. In this way, it allows for the identification of key parameters that have the most effect on the Defect fixing time and cost value so that further experimentation on these parameters can be performed and the parameters that have little effect can be ignored, as we explained in this paper.

2. Orthogonal Array Testing Strategy (OATS)

In order to overcome the challenges mentioned above, Orthogonal Array Testing Strategy (OATS) gives a systematic, statistical way of testing pair-wise interactions providing representative (uniformly distributed) coverage of all variable pair combinations. This makes the technique particularly useful for integration testing of software components.

It provides a representative (uniformly distributed) coverage of all variable pair combinations.

Pairwise (a.k.a. all-pairs) testing is an effective test case generation technique that is based on the observation that most faults are caused by interactions of at most two factors.

Pairwise-generated test suites cover all combinations of two and therefore are much smaller than exhaustive ones yet very effective in finding defects.

Dr. Genichi Taguchi was one of the first proponents of orthogonal arrays in test design. His techniques, known as Taguchi Methods, have been a mainstay in experimental design in manufacturing fields for decades.

The method of orthogonal arrays is an experimental design construction technique from the literature of

ISBN: 978-960-474-351-3

statistics. In turn, construction of such arrays depends on the theory of combinations. An orthogonal array is a balanced two-way classification scheme used to construct balanced experiments when it is not practical to test all possible combinations. The size and shape of the array depends on the number of parameters (factors) and values (levels) in the experiment. Orthogonal arrays are related to combinatorial designs.

Testing a software system requires the creation of test cases, which contain values for input parameters and the expected results. Exhaustive testing for all of the possible combinations of parameters, in most cases it is not possible, it is not feasible, or the cost is out of the available budget [4]. The main goal of using different methods and techniques of testing is to create a smaller number of combinations of parameters and their values, which will be tested.

According to inputting different combination of conditions so as to produce different impacts, software testing designs a large number of test cases. If the implementation of an overall test, due to the limit of the combination of conditions, it is difficult to carry out. In order to generate high quality test cases as early as possible to improve the efficiency of software testing, it is designed a generation tool of the automatic software testing case on orthogonal experimental design [5]. For the test data, the use of that tool design test cases. The practice shows that a small number of test cases are generated, the error detection ability is strong, and it greatly improves the efficiency of software testing.

In software testing process, it provides a natural mechanism for testing systems to be deployed on a variety of hardware & software configurations or with multiple interfaces. The combinatorial approach to software testing uses models to generate a minimal number of test inputs so that selected combinations of input values are covered. The most common coverage criteria are two-way or pair-wise coverage of value combinations, though for higher confidence three-way or higher coverage may be required.

The basic fault model that lies beneath existing techniques:

- Interactions and integrations are a major source of defects.

Most of these defects are a result of simple interactions such as in next example:

"When the background is blue and the font is Arial and the layout has menus on the right and the images are large and it's a Thursday then the tables don't line up properly."

- Most of these defects arise from simple pair-wise interactions such as in this error case:

"When the font is Arial and the menus are on the right, the tables don't line up properly."

- With so many possible combinations of components or settings, it is easy to miss one.
- Randomly selecting values to create all of the pair-wise combinations is bound to create inefficient test sets and test sets with random, less meaningful distribution of values.

OATS can be used to reduce the number of combinations and provide maximum coverage with a minimum number of test cases. OATS is an array of values in which each column represents a variable - factor that can take a certain set of values called levels.

- Orthogonal arrays are two dimensional arrays of numbers which possess the interesting quality that by choosing any two columns in the array you receive an even distribution of all the pair-wise combinations of values in the array.
- The size and shape of the array depend on the number of parameters and values in the experiment.
- Each row represents a test case/combination.
- In OATS, the factors are combined pair-wise rather than representing all possible combinations of factors and levels.

2.1 Orthogonal Array Testing Strategy applications

The OATS provides representative (uniformly distributed) coverage of all variable pair combinations. This makes the technique particularly useful for integration testing of software components (especially in OO systems where multiple subclasses can be substituted as the server for a client). It is also quite useful for testing combinations of configurable options (such as a web page that lets the user choose the font style, background color, and page layout). The size and shape of the array depend on the number of parameters and values in the experiment.

Definition 1: Orthogonal array $O(\rho, k, n, d)$

An orthogonal array is denoted by $O(\rho, k, n, d)$, where:

- ρ is the number of rows in the array. The k-tuple forming each row represents a single test configuration, and thus ρ represents the number of test configurations.
- k is the number of columns, representing the number of parameters.

ISBN: 978-960-474-351-3

- The entries in the array are the values 0, …, n – 1, where $n = f(n_0, \ldots, n_{k-1})$. Typically, this means that each parameter would have (up to) n values.
- d is the strength of the array (see below).

An orthogonal array has strength d if in any $\rho \times d$ sub-matrix (that is, select any d columns), each of the $n*d$ possible d-tuples (rows) appears the same number of times (>0). In other words, all d-interaction elements occur the same number of times.

Here is some terminology for working with orthogonal arrays followed by an example array in Table 1 [6,7]:

- **Runs** - ρ: the number of rows in the array. This directly translates to the number of test cases that will be generated by the OATS technique.
- **Factors** - k: the number of columns in an array. This directly translates to the maximum number of variables that can be handled by this array.
- **Levels** - n: the maximum number of values that can be taken on by any single factor. An orthogonal array will contain values from 0 to *Levels-1*.
- **Strength** - d: the number of columns it takes to see each of the $Levels^{Stength}$ possibilities equally often.
- Orthogonal arrays are most often named following the pattern $\mathbf{L_{Runs}(Levels^{Factors})}$.

Along with the more powerful software function and the improvement of software complexity, software development process is not easy to be controlled. Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and code generation. The design of tests for software and other engineered products can be as challenging as the initial design of the product itself. A rich variety of test case design methods have evolved for software.

The OATS makes the technique particularly useful for integration testing of software components. It is also quite useful for testing combinations of configurable options (such as a web page that lets the user choose the font style, background colour, and page layout). As an example of the benefit of using the OATS technique over a test set that exhaustively tests every combination of all variables, consider a system that has four options, each of which can have three values (see Table 1). The exhaustive test set would require 81 test cases (3 x 3 x 3 x 3 or the Cartesian product of the options). The test set created by OATS has only nine test cases, yet tests all of the pair-wise combinations. The OATS test set is only 11% as large at the exhaustive set and will uncover most of the interaction bugs. It covers 100% (9 of 9) of the pair-wise combinations, 33% (9 of 27) of the three-way combinations, and 11% (9 of 81) of the four-way combinations. What degree of interaction occurs in real system failures? Within the NASA database application, for example, 67 percent of the failures were triggered by only a single parameter value, 93 percent by two-way combinations, and 98 percent by three-way combinations. The detection-rate curves for the other applications studied are similar, reaching 100 percent detection with four- to six-way interactions. An orthogonal array is a balanced two-way classification scheme used to construct balanced experiments when it is not practical to test all possible combinations.

Table 1. *An $L_9(3^4)$ orthogonal array with 9 runs, 4 factors, 3 levels, and strength of 2*

Test Number	Factors			
	A	B	C	D
1	1	1	1	1
2	1	2	2	3
3	1	3	3	2
4	2	1	2	2
5	2	2	3	1
6	2	3	1	3
7	3	1	3	3
8	3	2	1	2
9	3	3	2	1

EXAMPLE 1: The above approach was used for a project where Compatibility testing had to be performed for various Browser-OS-Database combinations.

The factors and various levels for each of the factors are listed below in Table 2. The test set could easily be augmented if there were particularly suspicious three- and four-way combinations that should be tested. Interaction testing can offer significant savings.

ISBN: 978-960-474-351-3

Table 2. *Factors and Levels listed for the Compatibility Testing Scenario*

Factors	Level 1	Level 2	Level 3
A. Web Browser	IE 5.0	Netscape 4.7	Mozilla 1.3.1
B. Web Server OS	Sun OS 2.8	HP-UX 11	Windows NT Server 4.0
C. Type of Operation	Retrieval of Data	Saving of Data	Data Deletion
D. Database	Oracle 8i	SQL Server 7.0	Sybase ASE 12.5

Table 3. *Orthogonal Array constructed for the Compatibility Testing Scenario*

Test Number	Web Browser	Web Server OS	Type of Operation	Database
1	IE 5.0	Sun OS 2.8	Retrieval of Data	Oracle 8i
2	IE 5.0	HP-UX 11	Saving of Data	Sybase ASE 12.5
3	IE 5.0	Windows NT Server 4.0	Data Deletion	SQL Server 7.0
4	Netscape 4.7	Sun OS 2.8	Saving of Data	SQL Server 7.0
5	Netscape 4.7	HP-UX 11	Data Deletion	Oracle 8i
6	Netscape 4.7	Windows NT Server 4.0	Retrieval of Data	Sybase ASE 12.5
7	Mozilla 1.3.1	Sun OS 2.8	Data Deletion	Sybase ASE 12.5
8	Mozilla 1.3.1	HP-UX 11	Retrieval of Data	SQL Server 7.0
9	Mozilla 1.3.1	Windows NT Server 4.0	Saving of Data	Oracle 8i

Indeed a system with 20 factors and 5 levels each would require $\mathbf{5^{20} = 95\ 367\ 431\ 640\ 625}$ i.e. almost $\mathbf{10^{14}}$ exhaustive test configurations. Pair-wise interaction testing for $\mathbf{5^{20}}$ can be achieved in **45** tests. But what if some failure is triggered only by a very unusual combination of three, four, or more values? It's unlikely that our 45 tests would detect this unusual case. We would need to test at least three- and four-way value combinations. Combinatorial testing beyond pairwise is rare, however, because good algorithms for higher strength combinations haven't been available or were too slow for practical use. In the past few years, advances in covering-array algorithms, integrated with model checking or other testing approaches, have made it practical to extend combinatorial testing beyond pairwise tests [8]. If some failure is triggered only by an unusual combination of more than two factor interactions, how many testing combinations are enough to detect all errors? What degree of interaction occurs in real system failures? Surprisingly, researchers hadn't studied these questions when the US National Institute of Standards and Technology (NIST) began investigating causes of software failures in 1996 [8]. Study results showed that, across various domains, all failures could be triggered by a maximum of four- to six-way interactions. As Figure 1 shows, the detection rate increased rapidly with interaction strength. Within the NASA database application, for example, 67 percent of the failures were triggered by only a single parameter value, 93 percent by two-way combinations, and 98 percent by three-way combinations. The detection-rate curves for the other applications studied are similar, reaching 100 percent detection with four- to six-way interactions. These results are not conclusive, but they suggest that the degree of interaction involved in faults is relatively low, even though pairwise testing is insufficient. Testing all four- to six way combinations might therefore provide reasonably high assurance.

The OATS provides representative (uniformly distributed) coverage of all variable pair combinations. This makes the technique particularly useful for:

- integration testing of software components,
- testing combinations of configurable options (such as a web page that lets the user choose the font style, background colour, and page layout).

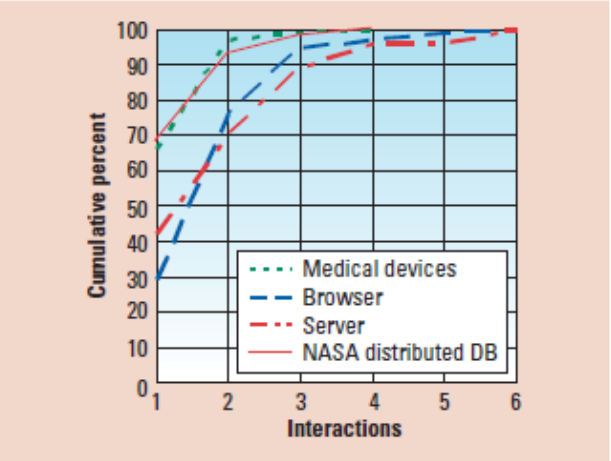

Figure 1. Error-detection rates for four- to six-way interactions in four application domains: medical devices, a Web browser, an HTTP server, and a NASA distributed database [8].

EXAMPLE 2: For *n* variables with *v* values, *k*-way combinations, Number of combinations for all combibnations is:

$$\rho_{Comb} = \binom{n}{k} \cdot v^k \qquad (1)$$

The OATS method provides much lower number of combinations for *k=2* way interaction, ie. pair-wise interaction of maximum No. of tests as:

$$\rho_{OATS} = n^2 + v_{\max} \log^2 v_{\max} \qquad (2)$$

ISBN: 978-960-474-351-3

In a specific example of a 12 variables: 7 Boolean, two 3-value, one 4-value, two 10-value in a typical test configuration for k-way interaction requires corresponding number of test combinations as shown in next Table:

k	# test cases
2-way	100
3-way	405
4-way	1,375
5-way	4,220
6-way	10,902

2.2 How to use this technique

The OATS technique is simple and straightforward. The steps are outlined below. The OATS technique is simple and straightforward. The steps are outlined below.

1. Decide how many independent variables will be tested for interaction. This will map to the **Factors** of the array.
2. Decide the maximum number of values that each independent variable will take on. This will map to the **Levels** of the array.
3. Find a suitable orthogonal array with the smallest number of **Runs**. A suitable array is one that has at least as many **Factors** as needed from Step 1 and has at least as many levels for each of those factors as decided in Step 2.
4. Map the **Factors** and values onto the array.
5. Choose values for any "left over" **Levels**.
6. Transcribe the **Runs** into test cases, adding any particularly suspicious combinations that aren't generated.

In a process of the combinatorial testing, tester generates tests that cover all double, triple or **n**-pairs combination of test parameters defined in the formal requirements for testing. Coverage of the pairs combination means that for any two parameters P1 and P2, and any valid values for the V1 for parameter P1 and V2 for parameter P2, there is a test in which the P1 has the value V1 and P2 has the value V2 [2,3]. Case studies [4,5,7,8] give evidence that the approach compared to conventional approaches is:

- more than twice as efficient (measured in terms of detected faults per testing effort) as traditional testing,
- about 20% more effective (measured in terms of detected faults per number of test cases) as traditional testing.

It is appropriate that the combinatorial testing uses orthogonal arrays and all-pairs algorithm for providing the following advantages [6]:

- Significantly reducing the cost and raises the quality of testing is achieved by intelligent generating test cases,
- Dramatically reduced overall number of test cases compared to exhaustive testing,
- Detects all faults due to a *single parameter* input domain,
- Detects all faults due to *interaction of two parameter* input domains,
- Detects many faults due to *interaction of multiple parameter* input domains

At this time, combinatorial testing is a very mature technique of testing, supported by a large number of tools to generate test cases [5].

3. The Software Defects Fixing Problem Optimization Using Taguchi Method - Case study

For over a century, Design of Experiments (DOE) methods have applied to testing in many areas such as medicine, chemistry, agriculture and manufacturing industry. Recently, the application of DOE has appeared in software testing. Exhaustive testing is not possible in most systems including software industry. Lot of time and effort in testing a product is put in covering all the different possible combinations. As the number of factors that decide the output of the system increase, the level of complexity involved in testing it also increases. Concepts like Orthogonal Arrays and ideas from the Design of Experiments can immensely improve software product testing even when the number of factors is large. DOE will help in reducing test cases and yet cover the input space efficiently.

Methods from the field of DOE have been applied to quality control problems in many engineering fields, including limited use for software [1,2,8,9,11], DOE seeks to maximize the amount of information gained in an experiment with an economical number of tests. Even a well-performed software process introduces defects that impact both development and customer systems. No matter how well we plan and shape software development, defects are generated and can escape to the customers. Failure to quickly resolve software problems leads to negative consequences for our customers and increases internal business costs. A quick deterministic

method to prioritize problems and implement their solution helps to reduce cycle time and costs [9].

Defects found in the later stages of the software life cycle tend to be harder to repair than those found in earlier stages [12]. Errors encountered by users are sometimes difficult to reproduce. To make matter worse, repairing one problem may introduce other problems into the system. The ability to predict defect repair time would be useful for creating testing plans and schedules, allocating resources and avoiding project overruns [3,4]. Estimated repair times can improve testing management and consequently, the reliability and time-to market of software.

Achieving this goal requires several steps [9]. The first is to determine a model that links problem resolution performance to institutional variables and problem characteristics. Statistical Design of Experiments (DOE) is a tool that provides data requirements for estimating the impacts of these variables on problem resolution. Once data has been gathered the results of statistical analysis can be input into a mathematical optimization model to guide the organization [11].

Our goal was to describe such an analysis. We used defect data published in [9] from previous software development efforts to describe the when-who-how approach for analyzing defect data to gain a better understanding of the quality control process and identify defect fixing problem improvement opportunities using Taguchi's Design of Experiments method. We used Analysis of Variance (ANOVA) to correlate problem resolution cycle time with three predictors, problem severity, problem complexity and engineer experience to find parametric equation for total software defect resolution time.

There were several benefits to the project:

- Optimal allocation of problems to the engineering staff resulted in savings of time and money.
- A closer relationship between experienced and novice engineers.
- Knowledge of the necessary problem resolution effort provided a baseline for further problem process improvement.

3.1 Statistically Designed Experiment

We needed to estimate the impact of the problem characteristics and engineer experience on problem resolution cycle time. Historical data was available, but inadequate. For example, the most complex problems had consistently been assigned to the most skilled engineers. Therefore we did not know how less experienced engineers would perform on difficult problems. This did not permit a full evaluation of our model. Statistical Design of Experiments is a set of techniques that help the analyst determine data requirements to estimate the parameters of any given model. There are several steps required to generate a statistically designed experiment:

1. Identify the response variable (or variables) to be modeled.
2. Determine the factors that influence that variable.
3. Determine the mathematical model by which the factors affect the response.
4. Determine appropriate factor settings.
5. Determine the number of "runs" required to estimate the model parameters. If possible, replicates are beneficial.

Step 1: *Response Variables*: Our goal was to understand problem resolution time. Two major components of cycle time were considered:

1. *Problem assessment time*.
2. *Problem resolution (implementation) time*.

These two sub-processes employ distinct procedures and are typically performed by different individuals. Until a problem has been properly assessed its true severity and complexity are poorly understood. Problem resolution cost was a second response and employed the same model as cycle time.

Step 2: *Factors*: The three predictor variables, problem severity, complexity and engineer experience were described above.

Step 3: *Model*: This required more thought. Some of the considerations are:

1. Do significant interactions occur? Gurus obviously have an advantage over novices for any type of problem. But perhaps that advantage is not constant. They may have an even greater advantage for some problems. This is an interaction effect.

2. Are there higher interactions? In this case a 3-factor interaction is the only higher order interaction available, given by Severity x Complexity x Experience. It was decided that such an interaction was highly unlikely.

The model that was selected included all possible 2-factor interactions. It is given by:

$$T = S + C + E + (S*C) + (S*E) + (C*E), \qquad (3)$$

where

ISBN: 978-960-474-351-3

T = Problem resolution time.

S = Problem Severity

C = Problem Complexity

E = Engineer Skill

Step 4: *Factor Levels*: The assigned factor values were:

- Problem Severity: Level 1 or **High,** Level 2 or **Low**
- Problem Complexity: Level 1 or **Simple,** Level 2 or **Involved** (**Average**) and Level 3 or **Complex**
- Engineer skill: Level 1 or **Novice,** Level 2 or **Experienced** and Level 3 or **Guru**

Step 5: *Number of Runs*: The product of the numbers of factor levels determines the total number of candidate experiments. With 2 x 3 x 3 = 18 runs one can estimate the model. In this paper we used sample of 18 data points from work [9]. The data are given in Table 1. Note that all possible combinations of factor levels are represented.

3.2 Statistical Method for Data Analysis

Analysis of Variance (ANOVA) was used to analyze closure time components. This is the standard method to correlate a numerical response with qualitative predictors. The full 2-factor interaction model discussed above was fit first. Any terms that failed statistical tests for significance were eliminated to avoid over-fitting the model. For both the assessment and implementation cycle time analyses the main effects of Complexity, Severity and Experience are significant at a very high confidence level (95%). We concluded that the variables to predict cycle time were chosen well.

Table 4. *Estimated Average Resolution Times and cost to fix*

Severity Level	Complexity Level	Engineer Experience	Assessment Mean [Days]	Implementation Mean [Days]	Total Resolution [Days]	Cost to Fix [$]
High	Simple	Novice	5.2	10.5	15.7	1177.5
		Experienced	3.9	9.0	12.9	1290
		Guru	3.4	6.4	9.8	1225
	Involved	Novice	5.3	11.3	16.6	1245
		Experienced	4.9	10.1	15.0	1500
		Guru	4.9	9.7	14.6	1825
	Complex	Novice	7.7	14.7	22.4	1680
		Experienced	6.9	14.3	21.2	2120
		Guru	5.2	10.5	15.7	1962.5
Low	Simple	Novice	2.4	6.3	8.7	652.5
		Experienced	2.1	2.5	4.6	460
		Guru	0.6	1.3	1.9	237.5
	Involved	Novice	5.2	10.4	15.6	1170
		Experienced	4.5	8.5	13.0	1300
		Guru	3.3	8.5	11.8	1475
	Complex	Novice	6.8	13.4	20.2	1515
		Experienced	5.1	11.1	16.2	1620
		Guru	4.6	8.9	13.5	1687.5

3.3 Mathematical Optimization Model

The optimization problem is to minimize total resolution cycle time by assigning a given set of problems, defined by severity and complexity, to engineers of three skill levels.

ISBN: 978-960-474-351-3

The linear programming method was used in work [9] to solve the optimization problem. There are 2x3x3 =18 decision variables that represent each possible type of assignment. Each represents a number of problems of given severity and complexity assigned to engineers with one of the three skill levels.

To make the notation more manageable the six classes of problems are indexed with numbers for problem complexity and engineer experience. The problems are labeled as follows:

H_1 = High Severity, Complexity 1 (simple) problems.

H_2 = High Severity, Complexity 2 (involved or average) problems.

H_3 = High Severity, Complexity 3 (complex) problems.

L_1 = Low Severity, Complexity 1 problems.

L_2 = Low Severity, Complexity 2 problems.

L_3 = Low Severity, Complexity 3 problems.

We also have the following resources.

E^1 = Number of Engineers of skill Level 1 (Novice), E^2 = Number of Engineers of skill Level 2 (Experienced), E^3 = Number of Engineers of skill Level 3 (Guru).

We denote the resource availability (in work days) by: T^1 = Time availability of novice engineers, T^2 = Time availability of experienced engineers, T^3 = Time availability of guru engineers.

Now we tie these definitions together. Let problems be indexed by severity i = H, L, complexity j=1,2,3, and engineer skill level k=1,2,3. Also identify the problem resolution average cycle times (estimated in the ANOVA) as:

t_{ij}^{k} = time required to resolve a problem of severity **i** and complexity **j** by an engineer of skill level **k**.

And the number of problems be identified as:

PR_{ij}^{k} = Total number of problems of severity **i** and complexity **j** assigned to engineers of skill level **k**.

The formal statement of the optimization problem given in problem report (PR) is then to Minimize total problem resolution time:

$$T = (t_{H1}^{1} \times PR_{H1}^{1}) + (t_{H1}^{2} \times PR_{H1}^{2}) + (t_{H1}^{3} \times PR_{H1}^{3}) + (t_{H21} \times PR_{H2}^{1}) + (t_{H2}^{2} \times PR_{H2}^{2}) + (t_{H2}^{3} \times PR_{H2}^{3}) + (t_{H3}^{1} \times PR_{H3}^{1}) + (t_{H3}^{2} \times PR_{H3}^{2}) + (t_{H3}^{3} \times PR_{H3}^{3}) + (t_{L1}^{1} \times PR_{L1}^{1}) + (t_{L1}^{2} \times PR_{L1}^{2}) + (t_{L1}^{3} \times PR_{L1}^{3}) + (t_{L2}^{1} \times PR_{L2}^{1}) + (t_{L2}^{2} \times PR_{L2}^{2}) + (t_{L2}^{3} \times PR_{L2}^{3}) + (t_{L3}^{1} \times PR_{L3}^{1}) + (t_{L3}^{2} \times PR_{L3}^{2}) + (t_{L3}^{3} \times PR_{L3}^{3}) \quad (4)$$

Subject to the following constraints.

1. The total number of problems of each severity / complexity class to be resolved.
2. The endowment of engineers of each skill level.
3. Each of the 18 decision variables must be non-negative.

These sets of constraints are given algebraically as follows.

1. There are six equality constraints for the total number of problems:

$$PR_{H1}^{1} + PR_{H1}^{2} + PR_{H1}^{3} = PR_{H1}$$

$$PR_{H2}^{1} + PR_{H2}^{2} + PR_{H2}^{3} = PR_{H2}$$

$$PR_{H3}^{1} + PR_{H3}^{2} + PR_{H3}^{3} = PR_{H3} \quad (5)$$

$$PR_{L1}^{1} + PR_{L1}^{2} + PR_{L1}^{3} = PR_{L1}$$

$$PR_{L2}^{1} + PR_{L2}^{2} + PR_{L2}^{3} = PR_{L2}$$

$$PR_{L3}^{1} + PR_{L3}^{2} + PR_{L3}^{3} = PR_{L3}$$

2. There are three constraints for the total number of engineer staff-days:

$$(T_{H1}^{1} \times PR_{H1}^{1}) + (T_{H2}^{1} \times PR_{H2}^{1}) + (T_{H3}^{1} \times PR_{H3}^{1}) + (T_{L1}^{1} \times PR_{L1}^{1}) + (T_{L2}^{1} \times PR_{L2}^{1}) + (T_{L3}^{1} \times PR_{L3}^{1}) \leq T^1$$

$$(T_{H1}^{2} \times PR_{H1}^{2}) + (T_{H2}^{2} \times PR_{H2}^{2}) + (T_{H3}^{2} \times PR_{H3}^{2}) + (T_{L1}^{2} \times PR_{L1}^{2}) + (T_{L2}^{2} \times PR_{L2}^{2}) + (T_{L3}^{2} \times PR_{L3}^{2}) \leq T^2 \quad (6)$$

$$(T_{H1}^{3} \times PR_{H1}^{2}) + (T_{H2}^{3} \times PR_{H2}^{3}) + (T_{H3}^{3} \times PR_{H3}^{3}) + (T_{L1}^{3} \times PR_{L1}^{3}) + (T_{L2}^{3} \times PR_{L2}^{3}) + (T_{L3}^{3} \times PR_{L3}^{3}) \leq T^3$$

3. There are 18 non-negativity constraints:

$$PR_{ij}^{k} \geq 0;\ i = L, H;\ j = 1, 2, 3;\ k = 1, 2, 3$$

3.4 Time Resolution Minimization Model

In the work [9] optimization was performed using an Excel add-in program to apply linear programming method that can be used to work out "what-if" scenarios, allowing project managers to see the consequences of choosing cost over schedule or vice-versa. The results appear in the following figure 2. Figure 2 shows the initial endowments of 900 novice, 450 experienced, and 250 guru staff-days [9]. There were 53 high severity problems, 16 of complexity level 1, 19 of level 2 and 18 of level 3. There were 22, 17, and 24 low severity problems of levels 1, 2, and 3, respectively. Assuming costs of $75, $100 and $125 per work day for the novice, experienced and guru skill levels, total problem resolution cost equals $139,145. The program indicates some slack resources i.e. 45 staff-days, 249.4 of the total allocation of 250 guru days were used. There were 9.2 days of experienced

ISBN: 978-960-474-351-3

engineer resources left and 35.1 days of novice resources. The optimal solution, for author in [9] is interesting because, contrary to expectations, the gurus are mostly assigned to low complexity problems, with some preference for low severity. The novices tended to be assigned to high severity, high complexity problems. This surprising result is due to the strong comparative advantage the gurus had in the less complex problems, requiring only two staff-days for the low severity ones!

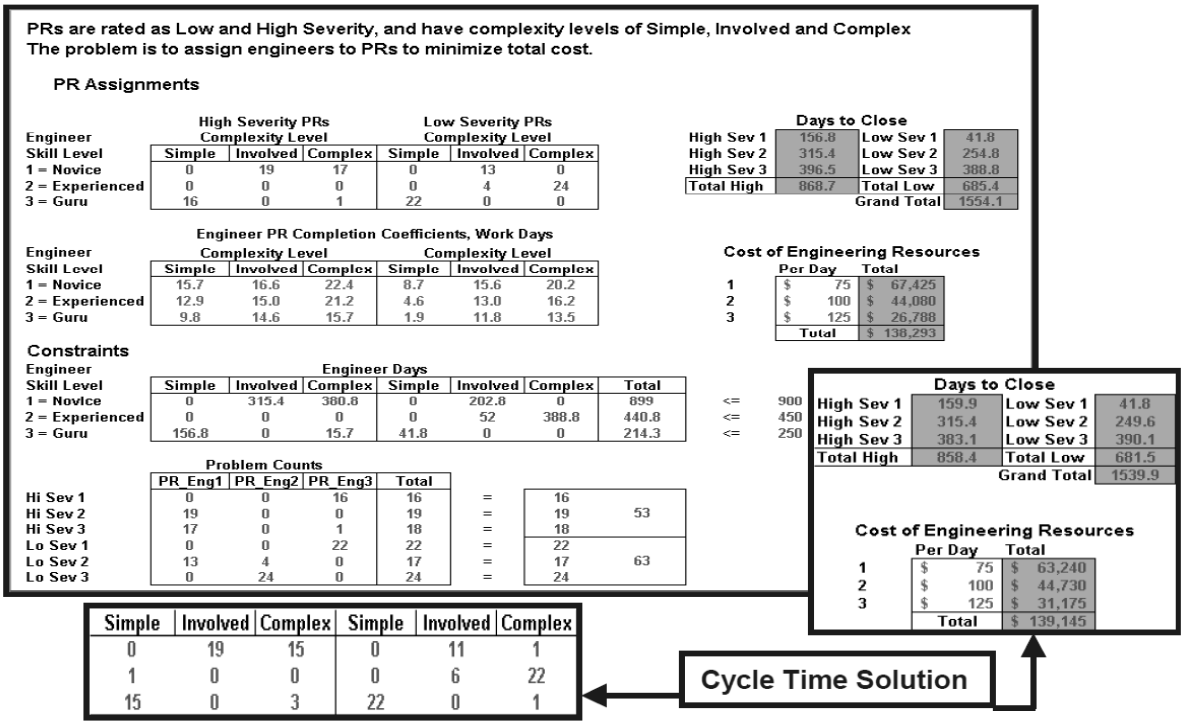

Figure 2. Output from the Time resolution and Cost Optimization Algorithm in [9]

3.5 Cost Minimization Model

The algorithm was run again to determine the cost minimizing solution [9]. The output from this exercise is found in figure 2. The total cost falls to $138,293 from the previous $139,145 for a savings of $852. The cycle time increases to 1,554.1 total days from 1,539.9 days for an increase of 14.2 days. This was accomplished by substituting some of the novice resources that had previously been slack (using 899 of their days compared to the previous 843.2 days). Guru days fell from 249.4 to 214.3, freeing up 35.1 days for them to work on other projects. 6.5 days of experienced engineers were also freed.

3.6 Optimizing Time resolution and cost to fix defects using Taguchi method

In our research we applied Taguchi screening designs for three controlled factors and levels of factors of each influence factors presented in Table 4. From 18 experiment runs for full-fractional design plan we used only total number of 9 treatments presented on Figure 3 (Excel sheet form). To analyze Time resolution and Cost to fix defects for MOTOROLA project data [9] we used MINITAB ver.16 statistical software tool. Some results are given in Fig. 3. The main effects plot and the intersection plot are useful tools for visualizing and analyzing the effects for factors. In this paper, we only use the main effects plot because we conclude from MINITAB 16 Taguchi experiment results, because factors interactions are not significant at 95% confidence level. The main effects plots for outputs: Resolution time and Cost are shown in Fig. 3.

High slope of line means that the factor gives more impact on the experimental results than other factors. Referring to the Ranking table of Taguchi analysis for Cost to fix [$] and Total Resolution time [Days] versus controlled factors Severity, Complexity and engineers Experience, the slope and graphs on figure 3, we observe that the most influential factor is the Complexity factor, then Severity, and Experience factor in our experiments for Cost to fix output. Also, we observe that the most influential factor for Total Resolution time output, again is the Complexity factor, then Experience, and Severity factor in our experiments. Taguchi optimization for cost to fix design explain author's [9] surprise that "The optimal solution is interesting because, contrary to expectations, the gurus are mostly assigned to low complexity problems, with some preference for low severity.".

ISBN: 978-960-474-351-3

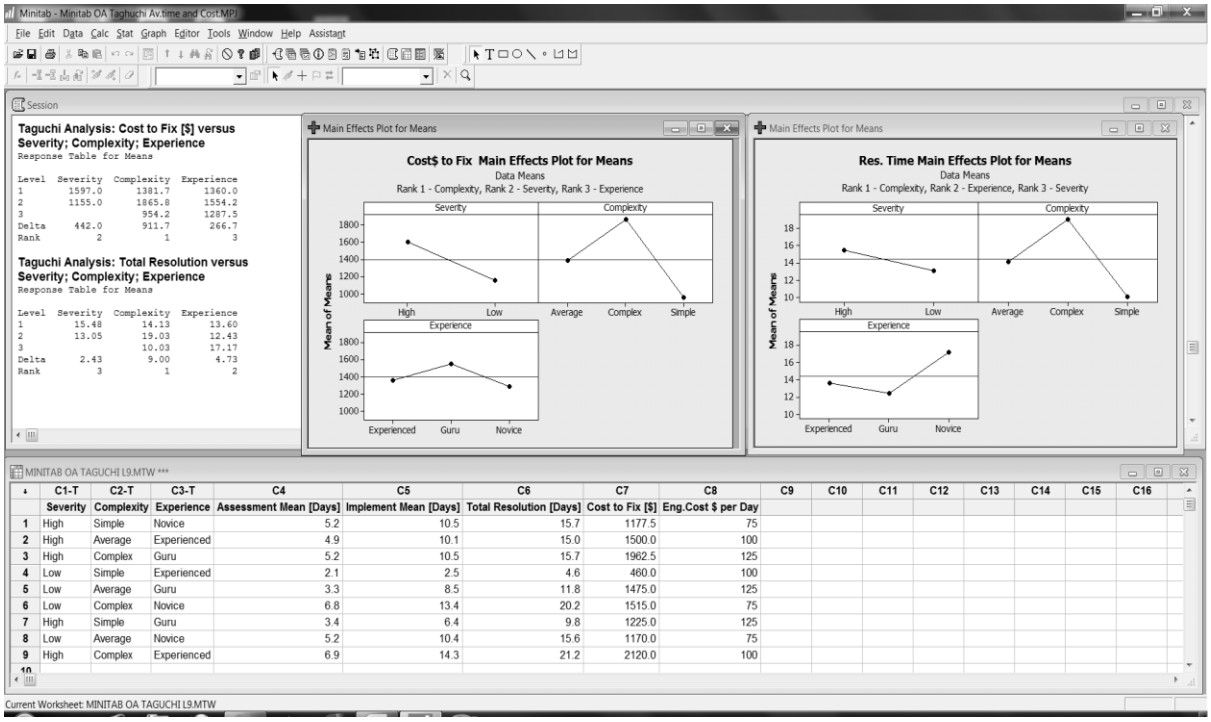

Figure 3. Output from the Time resolution and Cost Optimization using Taguchi method

According to Taguchi method for PR allocation we developed ranking procedure:

Step 1 - Assign PR of Simple complexity and Low severity (most influential factors first) to the Guru engineers, then

Step 2 - Simple complexity and High severity PR assign, again to Guru **if there exists** available time (days), then

Step 3 - Average complexity and Low severity PR, again assign to Guru **if there exists** available time (days) and to Experienced skill engineers, **OR if there does not** Guru available time (days) exists, then assign PR to Experienced skill engineers, then

Step 4 - Average complexity and High severity PR assign, again to Experienced skill engineers **if there exists** available time (days) and to Novice skill engineers, **OR if there does not exists** Experienced skill engineers available time (days), then assign PR to Novice skill engineers, and

Step 5 – Complex and Low, then Complex and High severity PR assign to the Novice until available time (days) exists.

Similar ranking procedure should be applied to optimize Total Resolution time taking into account that the most influential factor for Total Resolution time output, again is the Complexity factor, **then Experience**, and Severity factor in our experiments. Taghuchi approach optimization PR assignment tables for Total Resolution time and Cost to fix output are presented on Tables 5. Advantages of our Taguchi method application to minimize Total Resolution time output is that Total Res. Time=1550, %Delta= 0.7%, Delta=5 Days i.e. Total Resolution time is higher only 0.7%, compared to author's [9] linear programming approach, but in Taguchi case staff resource reserve is 50 days instead 45 days in author's [9] case.

Table 5. *Taguchi Problem Counts for Time resolution*

PR_Eng1	PR_Eng2	PR_Eng3	Sev * Compl.
0	16	0	Hi Sev 1
19	0	0	Hi Sev 2
18	0	0	Hi Sev 3
0	0	22	Lo Sev 1
0	13	4	Lo Sev 2
7	17	0	Lo Sev 3

Grand Total = $139665, %Delta= 0.4%, Instead of $138,293 Reserve 50 instead 45 Days Total Resolution Time 1550, %Delta= 0.7%, Delta=5 Days

This means that Taguchi approach can be used for accurate staff effort maintenance planning and, of course, for PR assignment effort for maintenance task

ISBN: 978-960-474-351-3

of delivered software product. Also, advantage of our Taguchi method application to minimize Cost to fix output is for Grand Total = $139,665 which is higher only 0.4%, compared to author's [9] linear programming approach equal to $138,293. Again, Taguchi approach can be used for more accurate staff effort maintenance planning and, of course, for PR assignment effort for maintenance task of delivered software product, allowing project managers to see the consequences of choosing cost over schedule or vice-versa.

4. Conclusion

Software testing need analysis and design test cases from different angles, and thus test the system effectively and scientifically. However, due to the limitation of the testing time and resources, it is impossible to test the system completely that is limited to test. That how to distribute limited resources to the system scientifically is an important topic in software testing. The Orthogonal experimental design is a good solution to such problems as we demonstrated in this paper using few examples from our research.

To select the right and strong representative points from a large number of test cases, which are more comprehensive and more objective understanding comprehensive tests, and to select the most optimal level combinations. This method can avoid one-sidedness and blindness testing, thereby can improve the efficiency of the software testing and can reduce the cost of it. Practice has proved that such orthogonal experimental design which is "equilibrium dispersion and neat comparable" is a kind of multi-factor testing and effective method. If it will put together a number of test cases designing techniques, the effect would be better.

Software defect repair time is an important factor in software development, and estimates of repair times are essential in planning, scheduling, and resource allocation in software projects. Repair time also depends on the experience and skills of the fixer, his or her workload, and other factors. Predicting defect repair time is a difficult problem that has not been researched as intensively as reliability modelling. The main goal is to gain a better understanding of the quality control process and identify defect fixing problem improvement opportunities using Taguchi's Design of Experiments method. We used Analysis of Variance (ANOVA) to correlate problem resolution cycle time with three predictors, problem severity, problem complexity and engineer experience to find parametric equation for total software defect cost and resolution time.

References

[1] Lj. Lazić S. Milinković, S. Ilić " OptimalSQM: Optimal Software Quality Management Repository is a Software Testing Center of Excellence", Proc. of 6th WSEAS European Computing Conference (ECC '12), Prague, Czech Republic, September 24-26, 2012, pp. 197- 209.

[2] D. R. Kuhn, N. Kacker, Yu Lei, "Practical Combinatorial Testing," NIST Special Publication Oct.2010.

[3] P. Flores and Y. Cheon, "Generating Test Cases for Pairwise Testing Using Genetic Algorithms," 18th IEEE International Symposium on Software Reliability Engineering (ISSRE'07), Dec.2007.

[4] Lj. Lazic and D. Velasevic, "Applying Simulation and Design of Experiments to the Embedded Software Testing Process," *Journal of Software Testing, Verification and Reliability*, Vol. 14, 2004, p.257–282

[5] http://www.pairwise.org/tools.asp

[6] Lj. Lazic, N. Mastorakis, "Orthogonal Array application for optimal combination of software defect detection techniques choices", WSEAS TRANSACTIONS on COMPUTERS, pp. 1319-1336, August 2008.

[7] J. Czerwonka, "Pairwise Testing in the Real World: Practical Extensions to Test-Case Scenarios", Microsoft Corporation, Software Testing Technical Articles, February 2008.

[8] D.R. Kuhn, Y.Lei, R. Kacker, "Practical Combinatorial Testing - Beyond Pairwise", IEEE IT Professional, June 2008.

[9] Porter D. Problem Resolution Optimization, Senior Statistician, Motorola, on web site www.stickyminds.com, visited 2013.

[10] Gopalakrishnan Nair, T R. Suma V., Nithya G. N. Estimation of the Characteristics of a Software Team for Implementing. Software Quality Professional; Mar 2011; 13, 2; ProQuest Central, pg. 14

[11] Lj. Lazić, I. Đokić, S. Milinković, „Estimating Cost and Defect Removal Effectiveness in SDLC Testing activities", INFOTEH-JAHORINA 2013, Jahorina, Proceedings Vol. 12,, ISBN 978-99955-763-1-8, March 2013. pp.572-577.

[12] Jones, Capers, Applied Software Measurement, Global Analysis of Productivity and Quality,Third Edition, New York: McGraw Hill, 2008.