A Greedy based Approach for Generating Minimal Covering Array and Optimal Test Suit for Combinatorial Testing

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Technology
in
Computer Science and Application

Submitted By
Ajay Chooramani
(Roll No. 601103002)

Under the supervision of
Mrs. Sunita Garhwal
Assistant Professor, SMCA

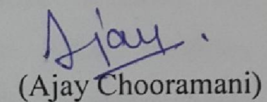
SCHOOL OF MATHEMATICS AND COMPUTER APPLICATION
THAPAR UNIVERSITY
PATIALA – 147004

July 2013

Certificate

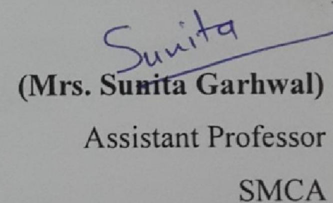
I hereby certify that the work which is being presented in the thesis entitled, **"A Greedy Based Approach for Generating Minimal Covering Array and Optimal Test Suit for Combinatorial Testing"**, in partial fulfillment of the requirements for the award of degree of Master of Technology in Computer Science and Applications submitted in School of Mathematics and Computer Applications, Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Mrs. Sunita Garhwal** and refers other researcher's work which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for award of any other degree of this or any other University.



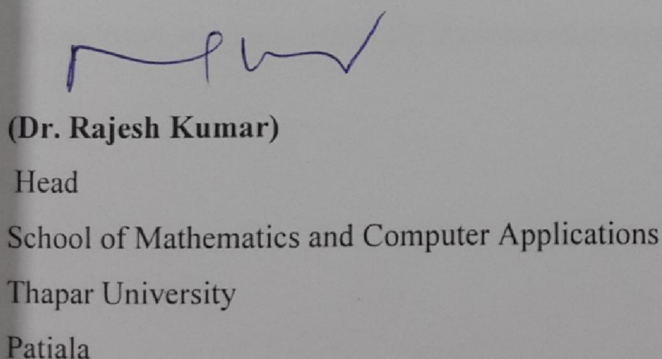
(Ajay Chooramani)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

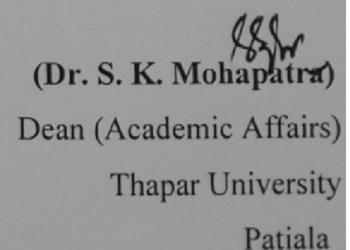


(Mrs. Sunita Garhwal)
Assistant Professor
SMCA

Countersigned by:



(Dr. Rajesh Kumar)
Head
School of Mathematics and Computer Applications
Thapar University
Patiala



(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

First of all I would like to thank the Almighty, who has always guided me to work on the right path of the life.

This work would not have been possible without the encouragement and able guidance of my supervisor **Mrs. Sunita Garhwal**. I thank my supervisor for their time, patience, discussions and valuable comments. Their enthusiasm and optimism made this experience both rewarding and enjoyable.

I am equally grateful to **Dr. Rajesh Kumar**, Associate Professor and Head, School of Mathematics and Computer Applications, for motivation and inspiration that triggered me for the thesis work.

I will be failing in my duty if I don't express my gratitude to **Dr. S. K. Mohapatra,** Senior Professor and Dean of Academic Affairs the University, for making provisions of infrastructure such as library facilities, computer labs equipped with net facilities, immensely useful for the learners to equip themselves with the latest in the field.

I am also thankful to the entire faculty and staff members of School of Mathematics and Computer Applications Department for their direct-indirect help, cooperation, love and affection, which made my stay at Thapar University memorable.

Last but not least, I would like to thank my parents for their wonderful love and encouragement, without their blessings none of this would have been possible. I would also like to thank my brother, since he insisted that I should do so. I would also like to thank my close friends for their constant support.

ABSTRACT

In software testing, a system have various factors like different configurations of hardware and software, or different types of input parameters and there values. If these factors have mutual interactions between them, and that may affect the software under test, then it is logical to test with a test suite covering all these factors and their interactions. But in such cases the necessary test suite is generally too large, making exhaustive testing usually impractical and often infeasible. As a result, we need to make a trade-off between testing efficiency and cost. One way to do this is to use Combinatorial Testing (CT), also called combinatorial interaction testing.

Combinatorial testing is applied for finding errors which are triggered by the interaction of parameters (configuration parameters and input parameters) of the software applications. Errors occur, when the usage of the software increases and interaction between those parameters grows rapidly.

Due to combinatorial explosion of values of parameters it is not possible to check all the possible combinations of values hence, pairwise testing provides an economical alternative to test all possible combinations of a set of variables/parameters. In pairwise testing a set of test cases is generated that covers all combinations of the selected test data values for each pair of variables.

Finding the least number of test cases has been proven to be an NP-complete problem .This means that an efficient way to find an optimal solution is not known and that the time required finding a minimum number of test cases grows rapidly when the numbers of parameters and possible values increase.

This thesis provides an algorithm and its implementation which tries to optimize the number of test case generated for combinatorial testing. All the results that have been obtained through this algorithm (applying on different number of parameters and there values) has been discussed at the end.

Table of Contents

Certificate *I*

Acknowledgment *II*

Abstract *III*

Table of Content *IV*

List of Figure *VI*

List of Tables *VII*

Abbreviations *VIII*

Chapter1: Combinatorial Testing

1.1 Combinatorial Testing 1

1.2 Some Basic Concepts to Understand Combinatorial Testing 1

1.2.1 Interaction Rule 1

1.2.2 Pairwise Testing 2

1.2.3 Notations 2

1.3 Pairwise Testing – A Shortcut for Combinatorial Testing 5

1.4 Types of Combinatorial Testing 7

1.4.1 Configuration Testing 7

1.4.2 Input Parameter Testing 8

1.5 Combinations are hard to Test 9

1.6 Advantages of Combinatorial Testing 10

1.7 Thesis Outline 11

Chapter 2: Literature survey

2.1 The State of CT Research 13

2.1.1. Model of SUT 13

2.1.2 Test Case Generation Technique: 15

2.1.3 Test Case Generation Tools 17

2.1.4. Test Case Prioritization 17

2.1.5 Failure Diagnosis 18
2.1.6 Metric and Evaluation 18
2.2 The Application of CT and Testing Procedure 18
Chapter 3: Problem Statement
3.1 Problem Statement 20
Chapter 4: Proposed Worked
4.1 Combination strategy for test case generation 22
4.1.1 Covering Array: 22
4.1.2 Seeding: 22
4.1.4 Methods to generate test suits: 24
4.2 Details of work done 25
4.2.1 Input given to the algorithm 26
4.2.2 Generation of output 27
4.2.3 Proposed Algorithm 28
4.3 Detail Proposed Algorithm 31
4.4 Example to show the working of algorithm 32
4.5 Results obtained 43
4.5.1 Result and Analysis 44
Chapter 5: Conclusion and Future Scope
5.1 Conclusion 46
5.2 Future Scope 45
REFERENCES 46
PUBLICATIONS 47

List of Figures

Figure 1.1:	Relation of Several Forms of CT And DOE	7
Figure 1.2:	System With 3 Inputs	8
Figure 1.3:	Types of Combinatorial Testing	10
Figure 1.4:	Advance Options Available In MS Word	13
Figure 2.1:	The Testing Procedure For CT	15
Figure 4.1:	Elements of Combination Strategy	26
Figure 4.2:	Different Types of Covering Array	26
Figure 4.3:	Types of Test Case Generation Approaches	28
Figure 4.4:	Overview of The Algorithm	30
Figure 4.5:	Modes of Input	30
Figure 4.6:	Manual Mode For Inserting Values	31
Figure 4.7:	Sample File In .Scv Format	31
Figure 4.8:	Generated Test Suits On An Excel Sheet	32
Figure 4.8:	Final Output Generated From Step 5	46
Figure 4.9:	Analysis of Algorithm	48

List of Tables

Table 1.1:	Configuration parameters for example 1.1	6
Table 1.2:	Test Case generated for pairwise testing for example 1.1	6
Table 1.3:	Test case generated for system with 3 parameters	9
Table 1.4:	4-Way Configuration testing	10
Table 1.5:	System's parameters and their type	11
Table 4.1:	All 2-way combinations for example 4.1	34
Table 4.2:	Initializing set A	37
Table 4.3:	Initializing List Bin step 2	39
Table 4.4:	Initializing Covering CA in step3	39
Table 4.5:	Add shipping parameter into CA step 4.1	39
Table 4.6:	List B after completion of step 4.1	41
Table 4.7:	Covering Array after adding uncovered pairs in rows	42
Table 4.8:	List B after completion of step 4.3	43
Table 4.9:	Covering CA after all parameters has been added	44
Table 4.10:	List B After all the parameters added into covering array	46
Table 4.11:	Results Obtained on various set of input	47

Abbreviation

CT	Combinatorial Testing
CA	Covering Array
SUT	Software Under Test
DOE	Design of Experiment
VCA	Variable Length Covering Array
NGS	Network Game Software
GA	Genetic Algorithm
ACA	Ant Colony Algorithm
IPO	In-order parameter Modeling
OATS	Orthogonal Array Testing Strategy

This chapter includes basic introduction of combinatorial testing and its type. It also consists, a brief introduction of the implementation of combinatorial testing, its advantages and its failure.

1.1 Combinatorial Testing

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 focused on identifying errors and faults, occurs due to the interaction of different parameters of software. Whenever we have a product that processes multiple variables that may interact with each other, we use combinatorial testing. The variables may come from a variety of sources, such the user interface, the different operating system, peripherals, different databases, or from across a network. The task in combinatorial testing goes beyond testing individual variables (although that must also be done, as well). In combinatorial testing the task is to verify that different combinations of variables are handled correctly by the system. Researchers have proposed and still are proposing new, effective and efficient techniques to refine the existing methods of testing and to find possible ways of improving the testing activity.

1.2 Some Basic Concepts of Combinatorial Testing

This section gives an overview of Combinatorial Testing, including some formal notations and definitions commonly used in this thesis.

1.2.1 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 [19].

1.2.2 Pairwise Testing

Pair-wise[12] testing 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. Pairwise testing is an economical alternative to testing all possible combinations of a set of variables. In pairwise testing a set of test cases is generated that covers all combinations of the selected test data values for each pair of variables. Pairwise testing is also referred to as all pairs testing and 2-way testing. It is also possible to do all triples (3-way) or all quadruples (4-way) testing, of course, but the size of the higher order test sets grows very rapidly.

1.2.3 Notations

We assume the Software under Test (SUT) has n parameters C_i (i = 1, 2… n), which may represent configuration parameters, internal or external events, user inputs, etc., and these parameters or their interactions may influence the SUT [27]. Let c_i have a_i discrete values from the finite set V_i, $a_i = |V_i|$. Assume all the parameters are independent, i.e., none of the parameter values is determined by the others. Without loss of generality, we assume $a_1 \geq a_2 \geq \ldots \geq a_n$. Let R be the set of interaction relations which records all the interactions existing among parameters. This information can be obtained from various development documents of the SUT or from interviews with related domain experts. $R \subseteq 2^c$, where C = $\{c_1, c_2 . . . c_n\}$. For example, given $R = \{\{c_1\}, \{c_1, c_2\}, \{c_2, c_3\}, \{c_3, c_4, c_5\}, \{c_4, c_5\}\}$. $\{c_1\} \in R$ shows that all the values of parameter c_1 in V_1 can affect the SUT and may trigger a software failure. $\{c_1, c_2\} \in R$ shows that there exist interactions between c_1 and c_2, that is, all the pairs in $V_1 \times V_2$ may trigger a software failure. $\{c_3, c_4, c_5\} \in R$ shows that there exist interactions among c_3, c_4, and c_5, and all the combinations in $V_3 \times V_4 \times V_5$ may affect the SUT. R can be viewed as the covering requirements for CT, specifying which combinations should be covered in testing. To present the key concepts of CT, we will make use of the following example:

Example 1.1: Consider a Software application which may be influenced by the configuration parameters as shown in Table 1.1. In we have n = 4, $a_1 = a_2 = a_3 = a_4 = 3$, $V_1 = \{a, b, c\}$, $V_2 = \{d, e, f\}$, $V_3 = \{1, 2, 3\}$, and $V_4 = \{4, 5, 6\}$. $R = \{\{c_1, c_2, c_3, c_4\}\}$, where $C = \{c_1, c_2, c_3, c_4\}$, that is, here all the parameters and their combinations may affect this application.

Table 1.1: Configuration parameters for example 1.1

Parameter1	Parameter2	Parameter3	Parameter4
A	D	1	4
B	E	2	5
C	F	3	6

Table 1.2: Test Case generated for pairwise testing for example 1.1

	C_1	C_2	C_3	C_4
T1	A	D	1	4
T2	A	E	2	5
T3	A	F	3	6
T4	B	D	2	6
T5	B	E	3	4
T6	B	F	1	5
T7	C	D	3	5
T8	C	E	1	6
T9	C	F	2	4

Definition 1: *Test Case* [25]: A test case t of SUT have n−tuples $(v_1, v_2 \ldots v_n)$ where $(v_1 \in V_1, v_2 \in V_2 \ldots, v_n \in V_n)$. Let T_{all} denote the set of all possible test cases for the SUT, that is, $T_{all} \subseteq \{(v_1, v_2 \ldots v_n) \mid (v_1 \in V_1, v_2 \in V_2 \ldots, v_n \in V_n)\} = V_1 \times V_2 \times \ldots \times V_n$. For any given SUT, there exists a T_{all} which is determined by the system parameters, their values and interactions, and some other constraints. For example, a test case t for above is the 4−tuple (A, D, 1, 4). Example 1.1 has a maximum of $3^4 = 81$ test cases based on all possible parameter value combinations. Obviously, T_{all} is generally too large to be completely executed in testing, and in fact it is not necessary to run all tests in T_{all}. In this article T_c will denote the test suite generated for CT, and obviously $T_c \subseteq T_{all}$.

The interaction between parameters is in fact, is the effect caused by the combination of the parameter values. When specific values of these parameters are used together, they may trigger a software failure. A combination of parameter values can be viewed as a value schema.

Definition 2: *Covering Array (CA)* [27]*:* If an N × n array has the following properties:

- Each column i ($1 \leq i \leq n$) contains only elements from the set V_i with $a_i = |V_i|$.
- The rows of each N × T subarray cover all T−tuples of values from the T columns at least once.

Then it is called a T−way covering array (or a Mixed strength T Covering Array (MCA)), denoted as CA (N; T, n, (a1, a2 . . . an)). The covering array number, CAN (N; T, n, (a1, a2. . . an)), is the minimum N required to satisfy the parameter T , n, and (a1, a2, . . . , an). When a1 = a2 = . . . = an = v, the covering array is written as CA (N; T, n, v), and its covering array number as CAN (N; T, n, v) [10, 11].

Testing with a test suite of T−way covering array is called T−way testing. T-way testing is a kind of CT which requires that every combination of any T parameter values in the software must be tested at least once. When T = 1, it is called the Each Choice (EC) combination strategy. It becomes Pairwise Testing (PW) when T = 2. When T = n, it is called the All Combination (AC) strategy. A more practical and in some cases more flexible approach to examining interaction coverage than T-way testing is the variable strength interaction testing [15]. Its test suite is called a variable strength covering array.

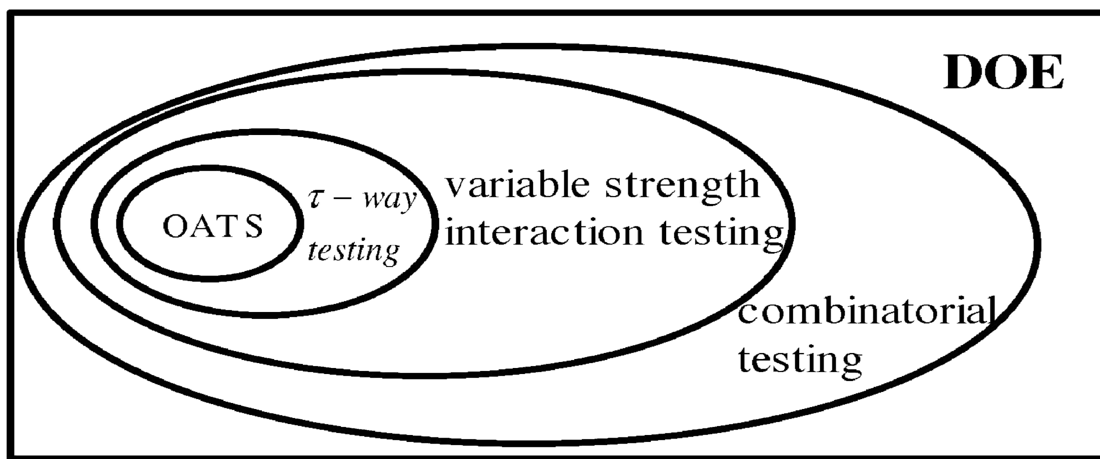


Figure 1.1: Relation of several forms of CT and DOE

This method uses a covering array that offers different covering strengths to different parameter groups and aims to cover only combinations of parameters having mutual interactions. This is important because interactions do not often exist uniformly between parameters. Some parameters will have strong interactions with each other while others may have few or no interactions. For this reason, we may wish to focus testing on a

specific set of parameters and apply a higher strength testing on them without ignoring the rest. In this case, the variable strength covering testing may be more effective and efficient.

Definition 3: *Variable Strength Covering Array* [27]: Let R denotes the set of the interaction relation set. R is the covering requirements for the SUT. Let CA be an $m \times n$ matrix, with the elements of column i of CA from V_i, which is the value set of parameter ci. If CA covers all the combinations required by R, we call CA a covering array for the SUT. Every row of CA is a test case. When R has elements of different sizes, that is, $R = \{\{c_{i1}, c_{i2}, \ldots, c_{iT}\} \mid 1 \leq i_1 < i_2 < \ldots < i_T \leq n, 1 \leq T \leq n\}$, CA is called a Variable strength Covering Array (VCA). When $R = \{\{c_{i1}, c_{i2} \ldots c_{iT}\} \mid \leq i_1 < i_2 < \ldots < i_T \leq n, T \text{ is fixed}\}$ and $|R| = {}^{T}_{n}C$, the covering array CA for the SUT is a T-way covering array as defined earlier.

Testing with a variable strength covering array is called variable strength interaction testing. T-way testing can be viewed as a special case of variable strength interaction testing, and both of them as special cases of CT. Actually CT can be expanded to include other important factors, such as constraints, and assigned test suite (seeds). CT can be viewed as a kind of Design of Experiment (DOE), which is a structured method that is used to determine the relationship between the different factors affecting a process and the output of that process [31]. For prioritized combinatorial testing, defines a new type of covering array, called ɭ -biased covering array [35]. A ɭ-biased covering array is a covering array CA (N; 2, k, v) in which the first ɭ rows form tests whose total benefit is as large as possible. That is, no CA (N'; 2, k, v) has ɭ rows that provide a larger total benefit. For instance, certain factors or levels for an input may have an associated benefit or priority that indicates a higher preference that the interaction be covered earlier in testing.

1.3 Pairwise Testing

Pairwise testing normally begins by selecting values for the system's input variables. These individual values are often selected using domain partitioning. The values are then permuted to achieve coverage of all the pairings. This is very tedious to do by hand. Practical techniques used to create pairwise test sets include Orthogonal Arrays. For a simple example of pairwise testing, consider the system in Figure 1.2. System S has three

input variables X, Y, and Z. Assume that set D, a set of test data values, has been selected for each of the input variables such that D(x) = {1, 2}, D(y) = {Q, R} and D(z) ={S, T}

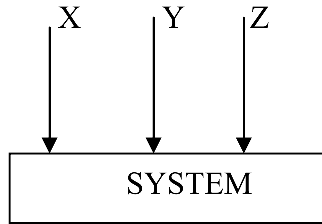


Figure 1.2: System with 3 inputs

The total number of possible test cases is 2×2×2 = 8 test cases. The pairwise test set has a size of only 4 test cases and is shown in Table 1.3.

Table 1.3: Test cases generated for system with 3 parameters

Test ID	**Input X**	**Input Y**	**Input Z**
TC1	1	Q	5
TC2	1	R	6
TC3	2	Q	6
TC4	2	R	5

In this example, the pairwise testing reduces test suit up to 50% from the full combinatorial test set of size 8. You can see from the table 1.3 that every pair of values is represented in at least one of the rows. If the number of variables and values per variable were to grow, the reduction in size of the test set would be more pronounced. For example a set of 10 variables with 2 possible values of each, makes 1024 test cases. But pairwise testing reduces these test cases to just 11 i.e. a reduction of 98%.

1.4 Types of Combinatorial Testing

Combinatorial testing is a vital approach to detect interaction errors occurs because of interaction of several parameters. There are two approaches [17] for combinatorial testing:

- Testing of configuration parameter values
- Testing of input parameter values

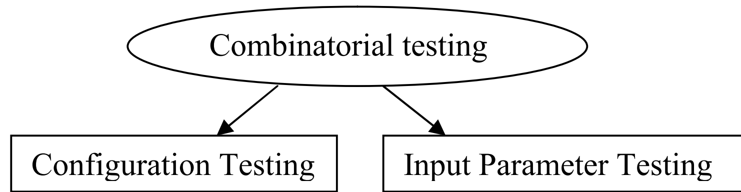


Figure 1.3: Types of combinatorial testing

1.4.1 Configuration Testing

In configuration testing [17], a system is tested against each possible configuration of software and hardware that are supported. Consider an application that is purporting on platforms comprised of the following five components:

Table 1.4: 4-Way Configuration Testing [3]

Test	OS	Browser	Protocol	CPU	DBMS
1	Windows 8	IE	IPv4	Intel	MySql
2	Windows 8	Firefox	IPv6	AMD	Sybase
3	Windows 8	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySql
5	OS X	IE	IPv4	Intel	Sybase
6	OS X	Firefox	IPv4	Intel	Oracle
7	RHEL	IE	IPv6	AMD	MySql
8	RHEL	Firefox	IPv4	Intel	Sybase
9	RHEL	Firefox	IPv4	AMD	Oracle
10	OS X	Firefox	IPv6	AMD	Oracle

An operating system (Windows 8, Apple OS X, Red Hat Enterprise Linux), browser (Internet Explorer, Firefox), protocol stack (IPv4, IPv6), processor (Intel, AMD), and database (MySql, Sybase, Oracle). Total number of possible platforms is equal to 3*2*2*2*3=72.

It is possible to check interaction of each component with other component using 10 test cases as shown in Table 1.4. Using these test cases, all possible pairs of platform components are covered.

1.4.2 Input Parameter Testing

In the Input Parameter testing [17], a system is tested against all the possible combination of values of parameters. Consider an on-line store that has four parameters of interest with following options:

- Three log-in types
- Three types of member status
- Three discount options
- Three shipping options.

Different end users may have different preferences and will probably to use different combinations of these parameters. For exhaustive testing, all combinations of the four parameters we need 81 test cases.

Table 1.5: System's parameters and their type [9]

Login type	Member status	Discount	Shipping
New member - not logged in	Guest	None	Standard (5-7 day)
New-member - logged in	Member	10% employee discount	Expedited (3-5 day)
Member - logged in	Employee	5% holiday discount	Overnight

In the online store, exhaustive testing requires 81 test cases, but pair-wise combinatorial testing uses only 9 test cases. Instead of testing each and every combination, all pairs of interactions between the parameters are tested. The resulting test suite is depicted in Table 1.5, which contain only 9 test cases.

Table 1.5: 2-Way Input-Parameter testing [9]

Test	Login type	Member status	Discount	Shipping
1	New member - not logged in	Guest	None	Standard (5-7 day)
2	New member - not logged in	Member	10% employee discount	Expedited (3-5 day)

3	New member - not logged in	Employee	$5offholiday discount	Overnight
4	New-member - logged in	Guest	$5 off holiday discount	Expedited (3-5 day)
5	New-member - logged in	Member	None	Overnight
6	New-member - logged in	Employee	10% employee discount	Standard (5-7 day)
7	Member - logged in	Guest	10% employee discount	Overnight
8	Member - logged in	Member	$5 off holiday discount	Standard (5-7 day)
9	Member - logged in	Employee	None	Expedited (3-5 day)

1.5 Combinatorial Explosion

Combinatorial Explosion describes the effect of functions that grow very rapidly as a result of combinatorial considerations. To be more clear on how the problem of combinatorial explosion could be resource and time consuming, consider for instance, the customize dialog in the tools menu of Microsoft Word as shown in Fig. 1.4. Even if only the toolbar tab is considered, there are 59 checkboxes to be tested. Therefore there are 2^{59} combinations of test cases to be evaluated. If the time required for one test case to be evaluated is 5 minutes, then it would require nearly 204280 years for a complete test of the toolbar tab alone. Therefore, it is very clear that combinatorial explosion is a serious issue which has to be considered and software testing always faces the problem of combinatorial explosion.

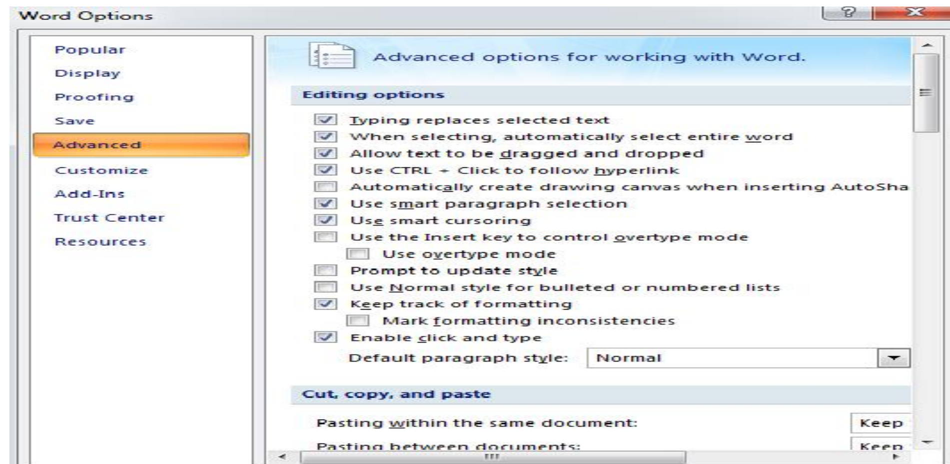


Figure 1.4: Advance options available in MS Word

Although it is important to test any software exhaustively, it is not practically possible to do so in reality owing to the cost and resources that are needed for the tests to be conducted. Therefore, one good solution is to construct a Test suite with an acceptable number of test cases for any t-way testing. There have been some solutions already proposed, however the problem of constructing the minimum test set for t-way testing is NP-complete and the challenges in this field still remain.

Combinatorial testing is difficult because of the large number of possible test cases. For example, in the web application like myntra or flipkart, we find a lot of categories to by a product. Suppose we have 10 different categories (size, color, brand, cost and so on) with 5 possible options available in it, the total of 5^{10}= 9765625 number of combinations are possible to be tested. Running all possible combinatorial test cases is generally not possible due to the large amount of time and resources required.

1.6 Advantages of Combinatorial Testing

Usages of combinatorial testing have a lot of advantages some of them are as follows:

- Combinatorial testing protects the system against pairwise or combinatorial bugs which occurs by the interaction between parameters.
- By applying combinatorial testing we can reduces the number tests to perform.
- Combinatorial testing is suitable for testing those applications have large number of parameters with large number of possible values.
- The availability of testing tools means, you no longer need to create these tests by hand.

1.7 Thesis Outline

This thesis is organized into five chapters. Chapter 1 describes about the basic concepts of Combinatorial Testing, a brief introduction to the implementation of combinatorial testing, its advantages and its failure. Chapter 2 describes information related to the history of Combinatorial Testing, literature survey which has been done during this thesis. Chapter 3 describes the motivation behind the thesis, discusses the problem statement and its objectives. Chapter 4 explains all the results obtain from the algorithm that has been developed for generating optimal test case suits. Chapter 5 summarizes the conclusions drawn from the work done along with the directions regarding the future work.

Chapter 2: Literature survey

Combinatorial Testing (CT) can detect failures that occur by interactions of parameters in the Software under Test (SUT). It has been an active field of research since last twenty years. This chapter aims to review previous work on CT, highlights the evolution of CT, and identifies important issues, methods, and applications of Combinatorial Testing.

2.1 Testing Procedures of Combinatorial Testing

In this section, we present the research topics of CT one by one. By adopting the work of others into the typical testing lifecycle, we propose a generic procedure model of CT, as shown in Figure 2.1. The first step is to build a model of SUT (Section 2.1.1), and the second step generates a test suite for CT (Section 2.1.2) and makes prioritization (Section 2.1.3). After the third step of test execution, if we find bugs and can fix them, then we can conduct regression testing else, if we cannot succeed in the failure diagnosis, we can handle some further testing (Section 2.1.4). Finally, we collect some data and evaluate the testing results (Section 2.1.5).

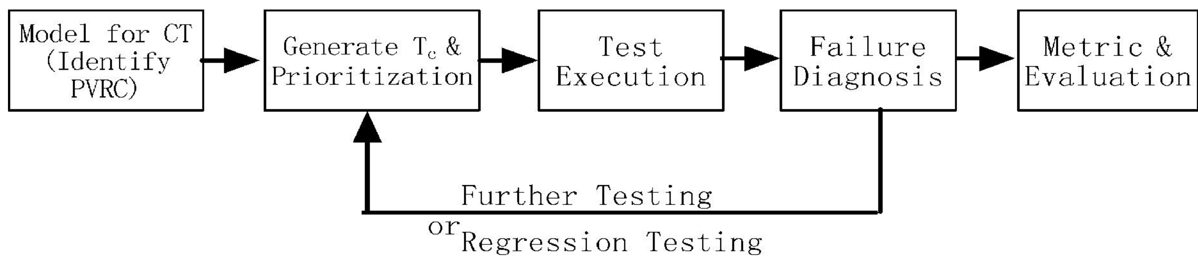


Figure 2.1: The testing procedures for CT [25]

Our testing procedure model is very similar to the common process of testing. However, to address the unique characteristics of CT, some special activities and techniques are added. We will discuss the application and the testing procedure of CT together (Section 2.1.6). Based on the generic model (Figure 2.1), we will discuss each step one by one. For the sake of clarity, we will present constraints and test generation together in Section 2.1.3, and metric and evaluation together in Section 2.1.6.

2.1.1 Model of SUT

Modeling of SUT is a very fundamental and important activity for CT, since the effectiveness and efficiency of CT depend on the model used. It is the starting point of

CT. Different testers may come up with different models of SUT based on their own skill and experience. A key issue in modeling is to build a precise model at the "right" level of abstraction. We first give the definition of model of SUT, then review some approaches for modeling a SUT.

Example 2.1: Take an example of a Network Game Software (NGS) which may be influenced by the configuration parameters. In our example, $n = 4$, $a1 = a2 = a3 = a4 = 3$, V_1 = {*Netscape, IE, Fire fox*}, V_2 = {*Windows, Linux, Macintosh*}, V_3 = {*ISDL, Modem, PN*}, and V_4 = {*Creative, Digital, Maya*}. $R = \{\{c_1, c_2, c_3, c_4\}\}$, where $C = \{ c_1, c_2, c_3, c_4\}$, that is, here all the parameters and their combinations may affect the NGS. We will take this example for the following discussion.

Definition 2.1: *SUT Mode:* For CT, a model of SUT includes four elements: parameters that may affect the SUT, values that should be selected for each parameter, interaction relations that exist between parameters and constraints that exist between values of the different parameters, which can be used to exclude combinations that are not meaningful from the domain semantics. A model for the SUT can be denoted as a 4-tuple: $Model_{SUT}$ (*P, V, R, C)*. By Definition 2.1, four key issues need to be resolved in modeling of SUT:

- How to identify parameters that may affect the SUT?
- What values should be selected for each parameter?
- What interactions exist between parameters?
- What constraints exist between parameters and their values?

Only when these issues are settled then CT can effectively brought into play. The first issue is to identify parameters for CT. Parameters in the SUT may represent configuration parameters, like those of the NGS in example 2.1, and similar cases found in Cohen et al. [10],Williams [37], and Xu et al. [41]. They may also represent user input parameters [32], features of the SUT [11], interfaces or GUI [7, 14], and components or objectives [12]. By reviewing various system documents and conducting some case studies [23], we can select an initial set of parameters for the SUT and then refine this set by adding or deleting parameters during the modeling procedure.

The second issue is to determine the applicable values for each parameter. This step is critical to the modeling of SUT, since the behavior of the SUT is governed by the specific combination of parameter values. Note that some parameters may have many values. For

example, the parameter “Audio” in the NGS, a configure parameter, can have many possible values, as there are hundreds of audio products in the market. When the parameter is an input of continuous values, it can have an infinite number of values. In this case, we must select some typical values. Equivalence partitioning, boundary value analysis, and primary element selection are the usual methods to be used here. The third issue is to identify the actual interactions between parameters. We can study the system documents to identify:

- Parameters which will not interact with any other parameters.
- Parameters which may have strong interactions with each other.
- Interactions which exist between a small numbers of parameters.

The fourth issue is to identify the constraints that exist between certain parameters. A common case occurs when some specific values of one parameter conflict with some values of another parameter. For example, the Browser parameter of the NGS cannot take on “Firefox” value when the Access parameter has ‘ISDL” value. This represents a kind of mutual exclusion constraint between parameters. The opposite case may also occur when some specific value of one parameter must be combined with some value of other parameters. For example, the Browser parameter of the NGS must take on “IE” value when the Access parameter has “ISDL” value or when the Audio parameter has “Maya” value. To obtain the information on the interactions and constraints between parameters, we can study the requirement document, design document, codes, and other related documents. We can also interview designers and programmers to extract the relevant information. Static analysis, such as program slicing, can also be used to identify the interactions between parameters as suggested by Schroeder and Korel [33, 34].

There have been many studies on modeling of SUT which helped to advance this field. For example, Dalal et al. [15, 16] learned that iteration and expert knowledge is required to build a proper model. Lott et al. [21] gave the samples of modeling system which can serve as a tutorial for applying CT. There have been some good guidelines for modeling of SUT. Many applicable scenarios have been given in Williams [41], and the procedure for determining the parameters and their values, and generating test suite is given in Krishnan et al. [27]. Mats and Offutt [23] presented a basic eight step process for input parameter modeling and gave some initial experience of using this process. Czerwonka [14] gave

some practical ways of modeling that make the pure pairwise testing approach more applicable.

Despite these prior works, there are still several open questions left in this area:

- How to effectively and efficiently model SUT for CT?
- How to validate the model?
- How to evaluate the effectiveness of the different approaches of modeling?

2.1.2 Test Case Generation Technique:

Test case generation is the most active area of CT research. As the problem of covering array generation is NP-hard, researchers have tried various methods to solve it. Till today, four main groups of methods have been proposed: greedy algorithm, heuristic search algorithm, mathematic method, and random method. The first two groups are computational approaches.

2.1.2.1 Greedy Algorithm

Greedy algorithms have been the most widely used method for test suite generation for CT. They construct a set of tests such that each test covers as many uncovered combinations as possible. There are two classes of greedy algorithms. The first class is the one-row-at-a-time variation based on the automatic test case generator AETG [11]. Bryce et al. [2] presented a generic framework for this class of algorithms. A single row of the array is constructed at each step until all T -sets have been covered. Algorithms that fit into this class include: the heuristic algorithm used to generate pairwise test suite of the CATS tool [28], the density-based greedy algorithm for generating a 2-way and higher strength covering array proposed by Bryce and Colbourn [2, 5, 6], and the greedy algorithm with different heuristics used in PICT in 2006. Tung and Aldiwan [36] also gave a greedy algorithm for a parametric test case generation tool that applies a combinatorial design approach to the selection of candidate test cases.

The second class of greedy algorithm is the In Parameter Order (IPO) algorithm. This class of algorithm begins by generating all T -sets for the first T factors and then incrementally expands the solution, both horizontally and vertically using heuristics until the array is complete.

2.1.2.2 Heuristic Search Based Algorithm

Heuristic search techniques such as hill climbing, great flood, tabu search, and simulated annealing have been applied to T-way covering array and VCA generation. Also some AI-

based search techniques such as Genetic Algorithm (GA) and Ant Colony Algorithm (ACA) have also been used. Heuristic search techniques start from a preexisting test set and then apply a series of transformations to the test set until it covers all the combinations. Ghazi [18] used a GA-based technique that identifies a set of test configurations that are expected to maximize pairwise coverage with a predefined number of test cases. Bryce and Colbourn [5] augmented the one-test-at-a-time greedy algorithm with a heuristic search, which can generate tests that have a high rate of T-tuple coverage. This approach combines the speed of greedy methods with the slower but more accurate heuristic search techniques. The hybrid approach seems to achieve a more rapid convergence of T-tuple coverage than either greedy or heuristic search alone. They compared four heuristic search techniques and found that hill climbing is effective only when time is severely constrained, but tabu search, simulated annealing, and the great flood perform better over a longer time. Cohen et al. [12] reported using the computational method of simulated annealing to generate 3-way covering array and variable strength array. Shiba et al. [29] also used the genetic algorithm and ant colony algorithm to generate the 3-way covering array. Heuristic search techniques can often produce a smaller test set than those from the greedy algorithm, but typically require a longer computation.

2.1.3 Test Case Generation Tools

More than 20 software tools have been developed for test case generation, for example, CATS (Constrained Array Test System) developed by Sherwood et al. [23], OATS (Orthogonal Array Test System) developed by Phadke et al. [6], AETG developed by Cohen et al. [11], IPO developed by Lei and Tai [27], T-config developed by Williams [38], CTS (Combinatorial Test Service) developed by IBM, and PICT developed by Microsoft [14]. Some new tools are still continually appearing. Many of these tools are freeware. As each tool has its own characteristics and advantages, none is the best for all settings. If possible we may use them together, and then choose the best result. Some integrated tools have been developed to generate a minimal test suite.

2.1.4. Test Case Prioritization

Test case prioritization has been well studied. The result of the prioritization is often a schedule of test cases so that those with the highest priority, according to some criterion,

are executed earlier in testing. When testing is terminated after running a subset of the prioritized test suite, those test cases deemed most important are executed. Especially when the resource is limited, the important test cases should be tested as early as possible. Well-ordered testing may reveal faults early because it ensures that the important test cases are executed first. The test case prioritization problem is defined as follows.

Definition 2.2: Given (T, Π, f), where T is a test suite, Π is the set of all test suites obtained by permuting the tests of T, and f is a function from Π to real numbers, the test case prioritization problem is to find $\pi \in \Pi$ such that $\forall \pi' \in, f(\pi) \geq f(\pi')$. Π gives the possible prioritization of T and f is the function to evaluate the prioritization as suggested by Bryce and Memon [4]. Prioritization can be computed in two ways: (1) reorder an existing test suite of CT based on prioritization criteria; and (2) generate an ordered test suite for CT, taking into account the importance of combinations. By Definition 2.2, the key issues in test prioritization are how to define the function f, and validate the effectiveness of the prioritization. Bryce and Colbourn [3] adapted a one-test-at-a-time greedy method to take importance of parameter pairs into account. With due consideration of seeding and constraint, they used this method to generate a set of tests in order. Bryce and Memon [4] explored the effectiveness of the prioritization with interaction coverage in the event-driven software. Compared with other criteria like ordering test by unique event coverage, ordering test by the length, and random ordering, prioritization with interaction coverage was found to provide the fastest rate of fault detection. Different prioritization methods have different performance. Better methods of test case prioritization for CT may be developed by considering additional factors. For example, we may do prioritization based on required execution time or based on incremental T-way coverage The prioritization done on GUI applications was also applied to Web application[30].We also need more empirical study to better understand the limitations and strengths of various proposed methods [20].

2.1.5 Failure Diagnosis

After detecting a failure, we need to investigate the failure to locate and then remove the fault. Kuhn and Reilly [20] suggested that software faults are often triggered by only a few interacting variables. Their results have important implications for CT. If all faults in SUT are triggered by a combination of no more than n parameters, then testing all n-way

combinations of parameters can provide a high confidence that nearly all faults have been discovered. So CT is an effective approach for detecting failures triggered by the combination of parameter values. When a defect is exposed in CT, we should determine which specific combination of parameter values causes the failure, or which value schema of SUT triggers the failure.

2.1.6 Metric and Evaluation

There exist two kinds of evaluation of CT that are

- Measurement and evaluation CT itself
- Measurement and evaluation of the quality of the SUT after CT.

CT has a natural metric, combination coverage, which measures the percentage of the covered parameter value combinations relative to the total combinations. One way to measure the combination coverage is based on the *k*-value schemas tested relative to the total *k*-value schemas.

2.2 Application of Combinatorial Testing

Since 1980 combinatorial testing has been applied in various fields and applications. For example, ADA compiler was first tested by combinatorial coverage in 1985 by R. Mandl [26]. Brownlie et al**.** tested PELMOREX (Weather forecast generator) by pairwise combinatorial coverage in 1992 and developed the OATS (Orthogonal Array Testing Strategy) system for test case generation. Many papers have been published that share the experience of using CT in practice. Huller [19] in year 2000 uses CT to test the ground system of satellite communication. Borroughs, Jain and et al. [1] shows in their paper that, how CT improves the quality and efficiency of internet protocol testing. Williams and Probert [40] explained in 1996 that how CT can be used to test network interface has been also used in other application too. White and Almezen [38] in year 2000 proposed a method for testing GUI objects. Empirical study of this method shows that, if we follow CT strategies then a less number of test cases can also detect the defects of the GUI. Burr and Young [7] generated test cases with the help of CT to test the email system with AETG. Configuration testing and browser testing were also evolved as the child of CT.

Chapter 3: Problem Statement

This chapter includes the problem statement and objectives of the thesis..

3.1 Problem Statement

Parameters and variables of computer software interact with each other to perform some operations. Sometimes this interaction between them creates bugs and errors in the system. For example, autopilot systems of airplanes have various parameter like pitch-altitude, altitude, airspeed, climb-rate, descent rate, roll-angle, angle-control-mode etc. These parameters have their own possible discrete values or a range of values. They often communicate with each other to automate the flying

A system with n parameters can have a *T-way* interaction level where T= 1, 2, 3…n. *2-way* interaction means 2 parameters of a system can interact or communicate with each other and focus is to find out all those bugs occur due to 2-way interaction

A complex automated system can have a number of parameters and a big range of possible values. Autopilot system has 19 parameters with 5 possible options each. A total of 5^{19} possible combinations exist. If human intervention is involved to run tests and verify results and if each test suite takes 15 minutes to run, then it will take roughly 24 staff-years to complete testing for an this system. With salary and benefit costs for each tester of $150,000, the cost of testing an app will be more than $3 million, making it virtually impossible to return a profit for most apps.

How can we provide effective testing for these types of typical systems at a reasonable cost? By developing an effective algorithm we can give an optimized or a reduced set of test cases. In above example of autopilot system, 94 test cases can cover all the possible 2-way combinations of configuration.

3.2 Thesis Objectives

Finding all 2-way combinatorial bugs from a software application using minimum number of test cases is a NP-Complete problem. Hence, there is always a possibility to develop an algorithm which gives more optimized test suit containing small number of test cases. Whole work of this thesis has been done to achieve by following objectives:

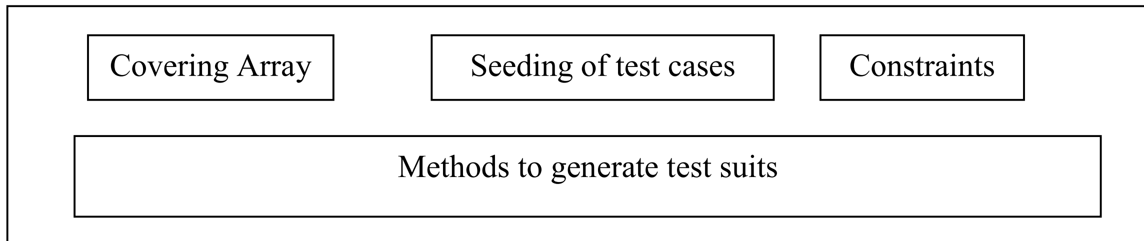
- To analyze various methodologies, techniques and algorithms used foe combinatorial testing
- To design a new algorithm which could give a more optimizes set of test cases to find all 2-way combinatorial bugs.
- To design a tool which implements this newly developed algorithm.
- To analyze and draw conclusions from various results obtained through this algorithm.

Chapter 4: Proposed Worked

This chapter contains the details of the work that has been done to meet all the thesis objectives.

4.1 Combination strategy for test case generation

A combination strategy selects test cases by combining values of different test object parameters based on some combinatorial strategy. It consist four elements as shown in figure 4.1:



4.1.1 Covering Array

There are many kinds of covering array we can select for CT. Figure 4.2 shows that different covering arrays have different sizes and offer different coverage. Based on the observation in testing that a larger test suite typically requires more testing cost and a higher coverage typically increases the chance of detecting failures, these covering arrays have different failure detection abilities and involve different testing costs.

Strategy	EC	BC(OFOT)	PW	OA	τ-way($\tau > 2$)	AC
Size	$Max_i(a_i)$	$\sum_{i=1}^{n} a_i - n + 1$	$\sim a_1^2$	$\sim a_1^2$	$\sim a_1 \cdots a_\tau$	$\prod a_i$
Coverage	$EC \leq BC$	$BC \leq PW$	$PW \leq OA$	$OA \leq \tau$-way	τ-way$\leq AC$	all

Figure 4.2: Different types of covering arrays[25]

4.1.2 Seeding

It means, to assign some specific test cases or some specific schema in testing. The seed tests are included in the generated test set without modification. The partial seed tests are seed tests that have some input fields which have unassigned values. The seed tests are included in the generated test set without modification. The partial seed tests are seed tests that have some input fields which have unassigned values. Seeding has following two practical applications:

- It allows explicit specification of important combinations. For example, if a tester is aware of combinations that are likely to be used in the field, he can specify a test

suite to contain these combinations. All T-way combinations in these seeds will be considered covered and only incremental test cases which contain the uncovered T-wise combinations will be generated and added.

- It can be used to minimize change in the test suite when the test domain description is modified and a new test suite regenerated. For example, a small modification of the test domain description, like adding parameters or parameter values, might cause big changes in the resulting test suite. Containing these changes can save testing cost.

4.1.3 Constraints

Constraints occur naturally in most systems. The typical situation is that some combinations of parameter values are invalid. Existence of constraints increases the difficulty in applying CT:

- Most existing test generation methods cannot deal with constraints, and ignore them. Ignoring constraints may lead to the generation of test configurations that are invalid. This can lead to ineffective test planning and wasted testing effort.
- It is difficult to design a general algorithm for test generation with due consideration of constraints.
- Even a small number of constraints may give rise to an enormous number of invalid configurations. When the generated test suite contains many invalid test cases, this will cause a loss of combination coverage.
- Complicated constraints may exist in SUT, and multiple constraints can interact to produce additional implicit constraints. It is both time consuming and highly error prone to deal with constraints manually in test suite generation. Thus, proper handling of constraints is a key issue we must address in test suite generation.

4.1.4 Methods to generate test suits

Test case generation is the most active area of CT research. As the problem of covering array generation is NP-hard, researchers have tried various methods to solve it. To date, four main groups of methods have been proposed: greedy algorithm, heuristic search

algorithm, mathematic method, and random method. The first two groups are computational approaches.

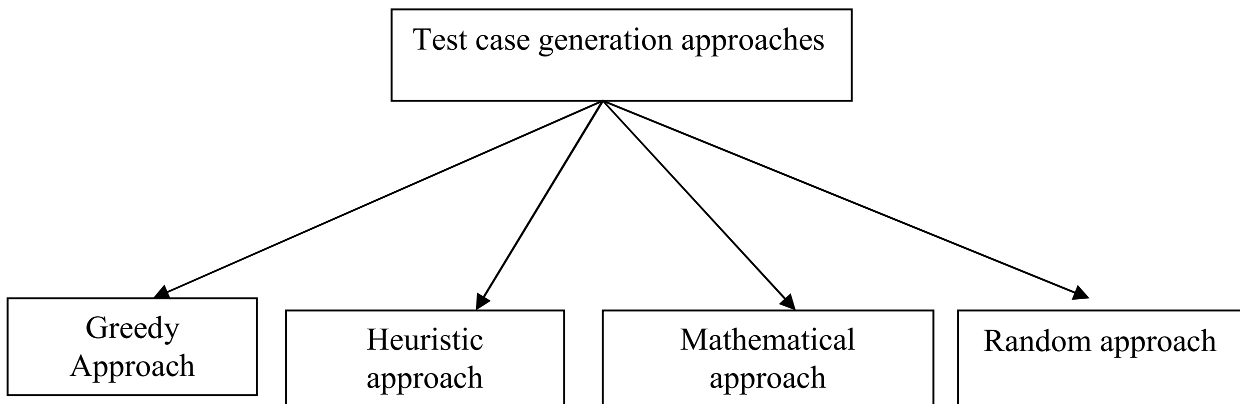


Figure 4.3 Different approaches for test case generation

4.1.4.1 Greedy Approach

Greedy algorithms construct a set of tests such that each test covers as many uncovered combinations as possible. A typical greedy algorithm works as follows.

Step1: Let U be the set of all the T-way combinations that should be covered according to the model of SUT (P, V, R, C).

step2: Let Seeds be the set of test cases assigned by testers. Remove all the T-way combinations covered by the test cases in Seeds from US.

Step3: While (U= Ø)

a) Generate a test case t to cover the most uncovered T-way combinations.
b) Check the constraints and ensure it is valid.
c) Remove all the T-way combinations covered by the test cases t from U.

End of while

4.1.4.2 Heuristic approach

Heuristic search techniques such as hill climbing, great flood, tabu search, and simulated annealing have been applied to *T*-way covering array and VCA generation. Also some AI-based search techniques such as Genetic Algorithm (GA) and Ant Colony Algorithm

(ACA) have also been used. We give a generic search algorithm for covering array generation:

Step 1: Let U be the set of all the T-way combinations that should be covered according to the model of SUT (P, V, R, C).

Step2: Let Seeds be the set of test cases assigned by testers. Remove all the T-way combinations covered by the test cases in Seeds from U.

Step 3: For each test case t, fitness (t) = the number of T-way combinations in U covered by t, but uncovered by the generated test cases and Seeds

Step 4: While (U= Ø)

a) Generate a set of test cases randomly;
b) Evolve the test set with a metaheuristic search method, such as simulated annealing, hill climbing, great flood, tabu search, particle swarm optimization, ant colony optimization, and genetic algorithm.
c) Output the valid t (satisfying the constraints) with the highest fitness score.
d) Remove all the T-way combinations covered by the test cases t from U.

End of while.

4.2 Preliminary work

We have tried to develop a tool which provides a covering array as an output, which consist of a lesser number of test cases. This tools works on our proposed algorithm. This algorithm has been developed by using greedy approach. This algorithm gives, a more optimized test suits to find all combinatorial bugs within a computer application. A set of parameters or variables and there values are send as an input into the tool. Algorithm gives a covering array, as the output. Finding an optimized covering array is a NP-Complete problem but we have tried to reduce our covering array as much as possible. Reduced array will give a lesser number of test cases for test suit.

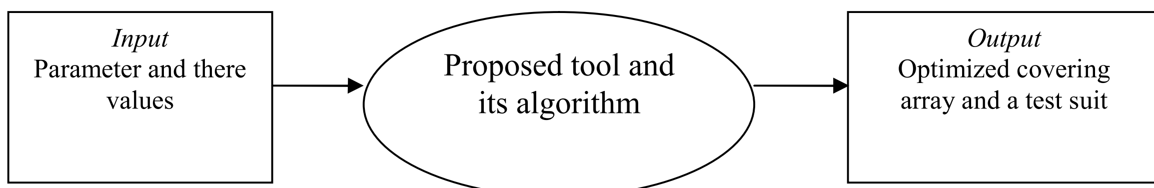


Figure 4.4: Overview of the algorithm

4.2.1 Input of the algorithm

To get a minimum covering array we send a set of parameters P_i where i=3, 4, 5, 6, 7, 8… n with their possible values in a set V_i where i=3, 4, 5, 6, 7, 8… n. We have assume that boundary value analysis and equivalence partitioning has been already been done on the set of parameters and the set of input we have is the refined values, that has to be tested. We have two modes to upload our value:

- Manual mode
- File upload in .csv format

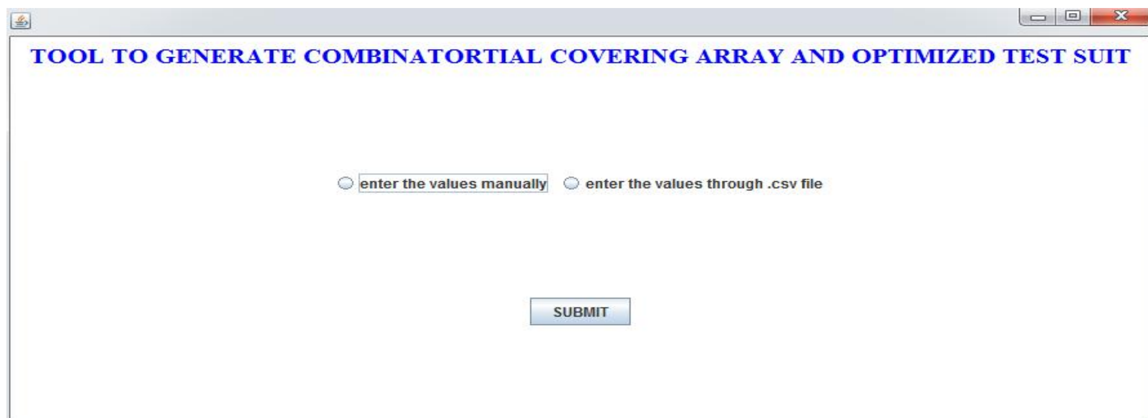


Figure 4.5: Modes of input

4.2.1.1 Manual mode

This mode ask users to enter the parameters and there values, manually. If we have a number of parameters with a big range of their possible values, then tool has the capability to enter these values for the parameters by its own, makes easy for user to insert the values. Following snapshot gives a view of the manual mode:

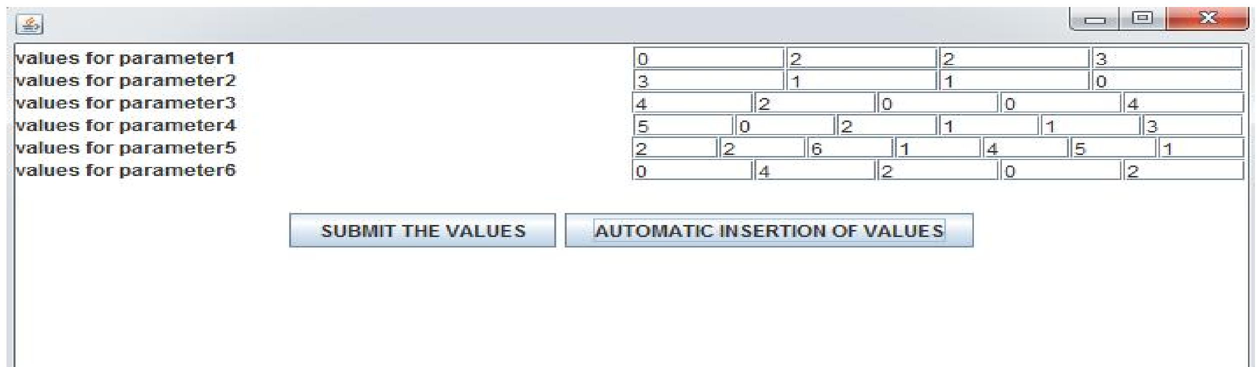


Figure 4.6: Manual mode for inserting values

4.2.1.2 File Upload in .csv Format

If an external user wants to test its application against 2-way combinatorial bugs, then, this is the other way to use this tool. User can send file in .csv format consisting of all the parameters and the values. This file can be uploaded into the tool to get optimized covering array. User need to send this file in an appropriate format. First line of the file should contain the number of parameters to be tested. After this, from second line till the end of file, each line consist of a parameter name and there values separated by comma. Figure 4.7 shows a sample of .csv format.

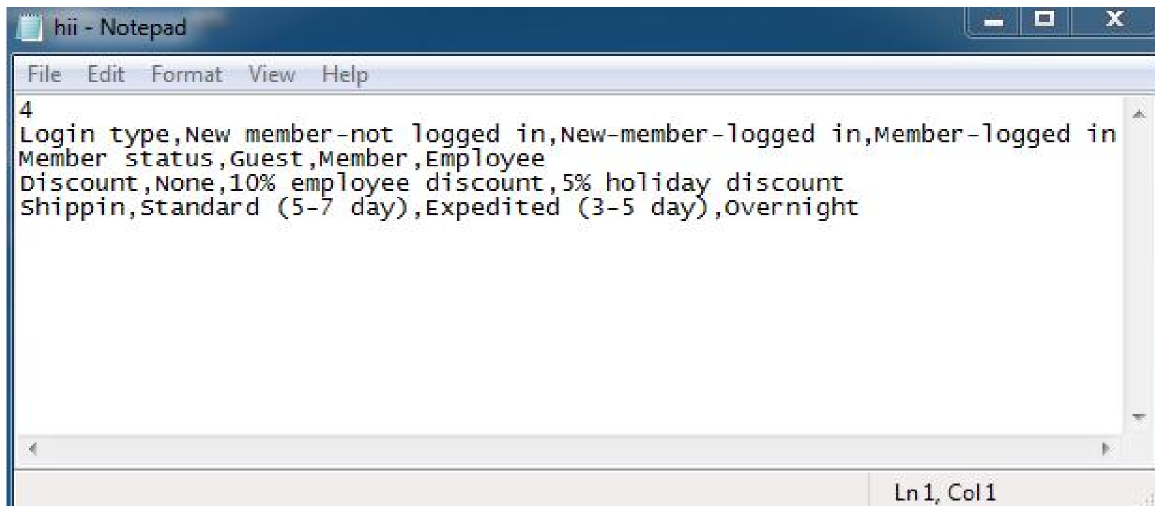


Figure 4.7: Sample file in .csv format

4.2.2 Generation of Output

Output consists of a test suit generated by this tool. Generated test suit contains, reduced number of test cases covering all 2-ways combinations of values. We get this test suit in the form of an excel sheet that can be used for further analysis. Whether the input mode is manual or through .csv file, we get a uniform output in excel format. Figure 4.8 gives a sample of output file.

	A	B	C	D	E
1	Test Case 1	New member-not logged in	Guest	None	Standard (5-7 day)
2	Test Case 2	New member-not logged in	Member	10% employee discount	Expedited (3-5 day)
3	Test Case 3	New member-not logged in	Employee	5% holiday discount	Overnight
4	Test Case 4	New-member-logged in	Guest	10% employee discount	Overnight
5	Test Case 5	New-member-logged in	Member	None	Standard (5-7 day)
6	Test Case 6	New-member-logged in	Employee	None	Expedited (3-5 day)
7	Test Case 7	Member-logged in	Guest	5% holiday discount	Expedited (3-5 day)
8	Test Case 8	Member-logged in	Member	None	Overnight
9	Test Case 9	Member-logged in	Employee	10% employee discount	Standard (5-7 day)
10	Test Case 10	New-member-logged in	Member	5% holiday discount	Standard (5-7 day)
11	Test Case 11	Member-logged in	Member	5% holiday discount	Standard (5-7 day)

Figure 4.8: Generated test suits on an excel sheet

4.2.3 Proposed Algorithm

This algorithm uses a greedy based approach for test suit generation. It gives a test suit with lesser number of test cases, covering all 2-way combination. Through this algorithm we have tried to reduce the NP-Complete problem of getting reduced number of test cases.

4.2.3.1 Formula generated for quick search

When user entered all the parameters and there values, a list is made by algorithm, containing all the 2-way possible combinations of values along there pair of parameters. To understand it lets take an example:

Example 4.1: We have four parameters A, B, C, D with (0, 1), (0, 1), (0, 1, 2), (0, 1, 2) values respectively. Total pairs of variables are {{AB, AC, AD}, {BC, BD}, {CD}} i.e. a total of 6 pairs of variables are possible.

A list is maintained by the algorithm containing all these pairs and their combined values as shown in Table 4.1.

Table 4.1: All 2-way combinations for example 4.1

Pair Name	Value 1	Value 2	Flag bit f
AB	0	0	uncovered
AB	0	1	uncovered
AB	1	0	uncovered
AB	1	1	uncovered
AC	0	0	uncovered
AC	0	1	uncovered
AC	0	2	uncovered
AC	1	0	uncovered
AC	1	1	uncovered
AC	1	2	uncovered
AD	0	0	uncovered
AD	0	1	uncovered
AD	0	2	uncovered
AD	1	0	uncovered
AD	1	1	uncovered
AD	1	2	uncovered
BC	0	0	uncovered
BC	0	1	uncovered
BC	0	2	uncovered
BC	1	0	uncovered
BC	1	1	uncovered
BC	1	2	uncovered
BD	0	0	uncovered
BD	0	1	uncovered
BD	0	2	uncovered
BD	1	0	uncovered
BD	1	1	uncovered
BD	1	2	uncovered
CD	0	0	uncovered
CD	0	1	uncovered
CD	0	2	uncovered
CD	1	0	uncovered
CD	1	1	uncovered

CD	1	2	uncovered
CD	2	0	uncovered
CD	2	1	uncovered
CD	2	2	uncovered

Quick search in combinatorial testing (Derivation of the formula)

Step1: Suppose we have n parameters, then, numbers of combinations for each parameter are as follows:

For parameter 1: n-1 number of combinations

For parameter 2: n-2 number of combinations

.
.
.
.

For parameter (n-1): 1combination

For parameter n: 0 combinations

Hence,

total possible combinations (tpc) are = (n-1) + (n-2) + (n-3) + … + 1 + 0

$$tpc = \frac{n(n-1)}{2} \quad (4.1)$$

Step2: If we want to see the total combinations between from k^{th} parameter to n^{th} parameter then,

$$\textit{Total combination from k to n (tckn)} = \frac{(n-k)(n-k+1)}{2} \quad (4.2)$$

Step3: If we want to get the index of the combination of kth and i^{th} parameter then, it is given by

Position = (tpc – tckn) + (i-k)

$$\text{Position} = \frac{(k-1)(2n-k)}{2} + i - k \quad (4.3)$$

So, eq4.3 is the formula to excess whole list of combinations quickly. We can directly reach to the position where all the combinations of kth and i^{th} parameter exist in the list.

Suppose for example 4.1we need to see the starting index of the combinations of parameter B and C in the list, using above formula (n=4, k=2, i=3)

$$Position = \frac{(2-1)(2*4-2)}{2} + 3 - 2$$

$$Position = 4$$

We can see into the table 4.1 that pair 4 consists of combinations for parameters B and C.

For calculating the pair index we have made a function calculate_index (n, k, i) that will return the index of the pair.

4.3 Proposed Algorithm

This section explains how this algorithm actually works. We are providing a pseudo code which explains the working of this algorithm. We have a set P containing input parameter where P_m is the mth parameter. We have a set A containing all the values of N parameters where A_{mn} is a set of values of parameter P_m where A_{mn} is the nth value for parameter m. List B contains all 2-way combinations of parameters in the form of tuple $\{T_{ij}, v_i, v_j, f\}$ where T_{ij} defines the pair-index of parameter i and parameter j whereas v_i and v_j are their possible values and f is the flag bit) telling this tuple is covered or not by the algorithm. We are also having a covering array CA initially with no element in it.

Input: Set of parameters and there possible values

Output: Optimized test suit

Algorithm: opt_test_Suit_CT

Step1: Initialize list B, set P and set A_{mn} with parameters and their values in it.

Step2: In list B, for each tuple with pair-index T_{12}, set flag f as covered.

Step3: Add all the tuples containing pair-index T_{12} from list B into CA.

//To add parameter m in covering array at column k

Step4: while all rows of CA not covered

Step4.1: Check for each data value (n) in A_{mn} for parameter P_m

Step 4.2: While all column j before column k, of row r are not accessed in CA, j starting

```
with 1
        4.2.1  q=calculate_index (N, k, j)
        4.2.2  while all pairs at pair-index q are not accessed
                    If pair ( j_value  , n) found at pair-index q in list B  then
                     Increase cover_counter for n^th value of parameter m by 1
                     Break;
              Else
                     Continue to access all the pairs at pair-index q.
               End  of Step 4.2.2
      j++;
End of  Step 4.2
n=n+1
```

Step 4.4: Repeat step4.1 till all n values of parameter m are not checked

Step 4.5: Get the value x of parameter m from set cover-counter for which we get the maximum Cover_counter value and add this value to column k of row r

```
            While all column l before column k, of row r are not accessed in   CA, l
            starting with 1
                   p =calculate_index (N, k , l)
                   found the location  of pair(l_value  , x) at pair-index p and set
                  f=covered .

            row++;
repeat step4 till all rows are not covered in CA.
End of Step 4
```

Step 5: In CA for column z to column k-1 starting with z=1

// **To add rows in CA with uncovered pairs from list B**

```
            W=calculate_index (N, z, k)
            if any pair has flag f =uncovered at pair-index W, then
                   Set f=covered ;
                   Create new row in CA and add values of this pair at
                   column z and column k respectively and Fill other
```

columns with their default values;

k = k+1;

i = i+1;

Step 6: Generate an excel sheet to display this covering array CA and test suit.

4.4 Numerical Example

To explain how this algorithm will work, let's take an example. Consider an on-line store that has four parameters of interest with following options:

- Three log-in types
- Three types of member status
- Three discount options
- Three shipping options.

Different end users may have different preferences and will probably to use different combinations of these parameters. For exhaustively testing, all combinations of the four parameters we need 81 test cases.

We will try to reduce the number of test cases with our proposed algorithm.

Step1: Set A is initialized with all the values of n parameters. Here we have 4 parameters with 3 possible values each.

Table 4.2: Initializing set A

Login type	**Member status**	**Discount**	**Shipping**
New member - not logged in	Guest	None	Standard (5-7 day)
New-member - logged in	Member	10% employee discount	Expedited (3-5 day)
Member - logged in	Employee	5% holiday discount	Overnight

Step2: List B contains all the possible pair values of all parameters with their pair-index. Here we have 6 pairs of parameters consisting 9 pairs each. Flag bit f for pair is initialized with "covered" as it is the pair between first two parameters.

Table 4.3: Initializing List B in step 2

Pair-index Parameters involved	Value1	Value2	Flag value
1 Login Type-Member Status	New member - not logged in	Guest	covered
	New member - not logged in	Member	covered
	New member - not logged in	Employee	covered
	New-member - logged in	Guest	covered
	New-member - logged in	Member	covered
	New-member - logged in	Employee	covered
	Member - logged in	Guest	covered
	Member - logged in	Member	covered
	Member - logged in	Employee	covered
2 Login Type-Discount	New member - not logged in	None	uncovered
	New member - not logged in	10% employee discount	uncovered
	New member - not logged in	5% holiday discount	uncovered
	New-member - logged in	None	uncovered
	New-member - logged in	10% employee discount	uncovered
	New-member - logged in	5% holiday discount	uncovered
	Member - logged in	None	uncovered
	Member - logged in	10% employee discount	uncovered
	Member - logged in	5% holiday discount	uncovered
3 Login Type-Shipping	New member - not logged in	Standard (5-7 day)	uncovered
	New member - not logged in	Expedited (3-5 day)	uncovered
	New member - not logged in	Overnight	uncovered
	New-member - logged in	Standard (5-7 day)	uncovered
	New-member - logged in	Expedited (3-5 day)	uncovered
	New-member - logged in	Overnight	uncovered
	Member - logged in	Standard (5-7 day)	uncovered
	Member - logged in	Expedited (3-5 day)	uncovered
	Member - logged in	Overnight	uncovered
	Guest	None	uncovered
	Guest	10% employee discount	uncovered
	Guest	5% holiday discount	uncovered
	Member	None	uncovered
	Member	10% employee discount	uncovered

4 Member Status-Discount	Member	5% holiday discount	uncovered
	Employee	None	uncovered
	Employee	10% employee discount	uncovered
	Employee	5% holiday discount	uncovered
5 Member Status-Shipping	Guest	Standard (5-7 day)	uncovered
	Guest	Expedited (3-5 day)	uncovered
	Guest	Overnight	uncovered
	Member	Standard (5-7 day)	uncovered
	Member	Expedited (3-5 day)	uncovered
	Member	Overnight	uncovered
	Employee	Standard (5-7 day)	uncovered
	Employee	Expedited (3-5 day)	uncovered
	Employee	Overnight	uncovered
6 Discount-Shipping	None	Standard (5-7 day)	uncovered
	None	Expedited (3-5 day)	uncovered
	None	Overnight	uncovered
	10% employee discount	Standard (5-7 day)	uncovered
	10% employee discount	Expedited (3-5 day)	uncovered
	10% employee discount	Overnight	uncovered
	5% holiday discount	Standard (5-7 day)	uncovered
	5% holiday discount	Expedited (3-5 day)	uncovered
	5% holiday discount	Overnight	uncovered

Step3: Copy all the values of pair I into the covering array.

Table 4.4: Initializing Covering Array CA in step 3

	Column1	**Column2**
Row1	New member - not logged in	Guest
Row2	New member - not logged in	Member
Row3	New member - not logged in	Employee
Row4	New-member - logged in	Guest
Row5	New-member - logged in	Member
Row6	New-member - logged in	Employee
Row7	Member - logged in	Guest
Row8	Member - logged in	Member
Row9	Member - logged in	Employee

Step 4.1: We follow the sub steps in Step 4.1 and get the table 4.4 as the new updated covering array. We have added new parameter “shipping” in column 3.

Table 4.5: CA after adding step 4.1(adding “shipping parameter”)

	Column1	Column2	**Column k=3**
Row1	New member – not logged in	Guest	**None**
Row2	New member – not logged in	Member	**10% employee discount**
Row3	New member – not logged in	Employee	**5% holiday discount**
Row4	New-member – logged in	Guest	**10% employee discount**
Row5	New-member – logged in	Member	**None**
Row6	New-member – logged in	Employee	**None**
Row7	Member – logged in	Guest	**5% holiday discount**
Row8	Member – logged in	Member	**None**
Row9	Member – logged in	Employee	**10% holiday discount**

All the pairs that has been covered in Covering Array are also get updated in list B by setting flag bit f as “covered” in respective pairs of list B. It has been shown in table4.4.

Table 4.6: List B after adding “Shipping” parameter in step 4.1

Pair-index **Parameters involved**	**Value1**	**Value2**	**Flag value**
1 Login Type-Member Status	~~New member - not logged in~~	~~Guest~~	covered
	~~New member - not logged in~~	~~Member~~	covered
	~~New member - not logged in~~	~~Employee~~	covered
	~~New-member - logged in~~	~~Guest~~	covered
	~~New-member - logged in~~	~~Member~~	covered
	~~New-member - logged in~~	~~Employee~~	covered
	~~Member - logged in~~	~~Guest~~	covered
	~~Member - logged in~~	~~Member~~	covered
	~~Member - logged in~~	~~Employee~~	covered
2 Login Type-Discount	~~New member - not logged in~~	~~None~~	covered
	~~New member - not logged in~~	~~10% employee discount~~	covered
	~~New member - not logged in~~	~~5% holiday discount~~	covered
	~~New-member - logged in~~	~~None~~	covered
	~~New-member - logged in~~	~~10% employee discount~~	covered
	New-member - logged in	5% holiday discount	uncovered
	~~Member - logged in~~	~~None~~	covered

	~~Member - logged in~~	~~10% employee discount~~	covered
	~~Member - logged in~~	~~5% holiday discount~~	covered
3 Login Type-Shipping	New member - not logged in	Standard (5-7 day)	uncovered
	New member - not logged in	Expedited (3-5 day)	uncovered
	New member - not logged in	Overnight	uncovered
	New-member - logged in	Standard (5-7 day)	uncovered
	New-member - logged in	Expedited (3-5 day)	uncovered
	New-member - logged in	Overnight	uncovered
	Member - logged in	Standard (5-7 day)	uncovered
	Member - logged in	Expedited (3-5 day)	uncovered
	Member - logged in	Overnight	uncovered
4 Member Status-Discount	~~Guest~~	~~None~~	covered
	~~Guest~~	~~10% employee discount~~	covered
	~~Guest~~	~~5% holiday discount~~	covered
	~~Member~~	~~None~~	covered
	~~Member~~	~~10% employee discount~~	covered
	Member	5% holiday discount	uncovered
	~~Employee~~	~~None~~	covered
	~~Employee~~	~~10% employee discount~~	covered
	~~Employee~~	~~5% holiday discount~~	covered
5 Member Status-Shipping	Guest	Standard (5-7 day)	uncovered
	Guest	Expedited (3-5 day)	uncovered
	Guest	Overnight	uncovered
	Member	Standard (5-7 day)	uncovered
	Member	Expedited (3-5 day)	uncovered
	Member	Overnight	uncovered
	Employee	Standard (5-7 day)	uncovered
	Employee	Expedited (3-5 day)	uncovered
	Employee	Overnight	uncovered
6 Discount-Shipping	None	Standard (5-7 day)	uncovered
	None	Expedited (3-5 day)	uncovered
	None	Overnight	uncovered
	10% employee discount	Standard (5-7 day)	uncovered
	10% employee discount	Expedited (3-5 day)	uncovered
	10% employee discount	Overnight	uncovered
	5% holiday discount	Standard (5-7 day)	uncovered

	5% holiday discount	Expedited (3-5 day)	uncovered
	5% holiday discount	Overnight	uncovered

Step 4.2: All the pair- indexes that have been used till now are scanned again to get remaining uncovered pairs in them. Sub steps of Step 4.2 are applied to add new rows to cover these uncovered pairs. Covering Array CA obtained after the completion of Step 4.2 is shown in table 4.7.

Table 4.7: Covering Array CA after adding uncovered pairs in rows

	Column1	**Column2**	**Column k=3**
Row1	New member - not logged in	Guest	**None**
Row2	New member - not logged in	Member	**10% employee discount**
Row3	New member - not logged in	Employee	**5% holiday discount**
Row4	New-member - logged in	Guest	**10% employee discount**
Row5	New-member - logged in	Member	**None**
Row6	New-member - logged in	Employee	**None**
Row7	Member - logged in	Guest	**5% holiday discount**
Row8	Member - logged in	Member	**None**
Row9	Member - logged in	Employee	**10% holiday discount**
Row10	***New member - not logged in***	***Employee***	***5% holiday discount***
Row11	***New member - not logged in***	***Member***	***5% holiday discount***

Status of list B has been shown in table 4.7 after adding a column and rows into CA.

Table 4.8: List B after completion of step 4.2

Pair-index Parameters involved	**Value1**	**Value2**	**Flag value**
1 Login Type-Member Status	~~New member - not logged in~~	~~Guest~~	covered
	~~New member - not logged in~~	~~Member~~	covered
	~~New member - not logged in~~	~~Employee~~	covered
	~~New-member - logged in~~	~~Guest~~	covered
	~~New-member - logged in~~	~~Member~~	covered
	~~New-member - logged in~~	~~Employee~~	covered
	~~Member - logged in~~	~~Guest~~	covered
	~~Member - logged in~~	~~Member~~	covered
	~~Member - logged in~~	~~Employee~~	covered
2	~~New member - not logged in~~	~~None~~	covered

Login Type-Discount	~~New member - not logged in~~	~~10% employee discount~~	covered
	~~New member - not logged in~~	~~5% holiday discount~~	covered
	~~New-member - logged in~~	~~None~~	covered
	~~New-member - logged in~~	~~10% employee discount~~	covered
	~~New-member - logged in~~	**~~5% holiday discount~~**	**covered**
	~~Member - logged in~~	~~None~~	covered
	~~Member - logged in~~	~~10% employee discount~~	covered
	~~Member - logged in~~	~~5% holiday discount~~	covered
3	New member - not logged in	Standard (5-7 day)	uncovered
Login Type-Shipping	New member - not logged in	Expedited (3-5 day)	uncovered
	New member - not logged in	Overnight	uncovered
	New-member - logged in	Standard (5-7 day)	uncovered
	New-member - logged in	Expedited (3-5 day)	uncovered
	New-member - logged in	Overnight	uncovered
	Member - logged in	Standard (5-7 day)	uncovered
	Member - logged in	Expedited (3-5 day)	uncovered
	Member - logged in	Overnight	uncovered
4	~~Guest~~	~~None~~	covered
Member Status-Discount	~~Guest~~	~~10% employee discount~~	covered
	~~Guest~~	~~5% holiday discount~~	covered
	~~Member~~	~~None~~	covered
	~~Member~~	~~10% employee discount~~	covered
	~~Member~~	**~~5% holiday discount~~**	**covered**
	~~Employee~~	~~None~~	covered
	~~Employee~~	~~10% employee discount~~	covered
	~~Employee~~	~~5% holiday discount~~	covered
5	Guest	Standard (5-7 day)	uncovered
Member Status-Shipping	Guest	Expedited (3-5 day)	uncovered
	Guest	Overnight	uncovered
	Member	Standard (5-7 day)	uncovered
	Member	Expedited (3-5 day)	uncovered
	Member	Overnight	uncovered
	Employee	Standard (5-7 day)	uncovered
	Employee	Expedited (3-5 day)	uncovered
	Employee	Overnight	uncovered
6	None	Standard (5-7 day)	uncovered

Discount-Shipping	None	Expedited (3-5 day)	uncovered
	None	Overnight	uncovered
	10% employee discount	Standard (5-7 day)	uncovered
	10% employee discount	Expedited (3-5 day)	uncovered
	10% employee discount	Overnight	uncovered
	5% holiday discount	Standard (5-7 day)	uncovered
	5% holiday discount	Expedited (3-5 day)	uncovered
	5% holiday discount	Overnight	uncovered

Covering CA after all parameters has been added

Table 4.9 shows the final covering array generated after insertion of third and forth parameter

Table 4.9: Covering CA after all parameters has been added

	Column1	**Column2**	**Column k=3**	**Column k=4**
Row1	New member - not logged in	Guest	**None**	**Standard (5-7)**
Row2	New member - not logged in	Member	**10% employee discount**	**Expedited (3-5)**
Row3	New member - not logged in	Employee	**5% holiday discount**	**Overnight**
Row4	New-member - logged in	Guest	**10% employee discount**	**Overnight**
Row5	New-member - logged in	Member	**None**	**Standard (5-7)**
Row6	New-member - logged in	Employee	**None**	**Expedited (3-5)**
Row7	Member - logged in	Guest	**5% holiday discount**	**Expedited (3-5)**
Row8	Member - logged in	Member	**None**	**Overnight**
Row9	Member - logged in	Employee	**10% holiday discount**	**Standard (5-7)**
Row10	***New member - not logged in***	***Employee***	***5% holiday discount***	**Standard (5-7)**
Row11	***New member - not logged in***	***Member***	***5% holiday discount***	**Standard (5-7)**

Table 4.10: List B After all the parameters added into covering array

Pair-index Parameters involved	**Value1**	**Value2**	**Flag value**
1	~~New member - not logged in~~	~~Guest~~	covered
	~~New member - not logged in~~	~~Member~~	covered

Login Type-Member Status	~~New member - not logged in~~	~~Employee~~	covered
	~~New member - logged in~~	~~Guest~~	covered
	~~New member - logged in~~	~~Member~~	covered
	~~New member - logged in~~	~~Employee~~	covered
	~~Member - logged in~~	~~Guest~~	covered
	~~Member - logged in~~	~~Member~~	covered
	~~Member - logged in~~	~~Employee~~	covered
2 Login Type-Discount	~~New member - not logged in~~	~~None~~	covered
	~~New member - not logged in~~	~~10% employee discount~~	covered
	~~New member - not logged in~~	~~5% holiday discount~~	covered
	~~New member - logged in~~	~~None~~	covered
	~~New member - logged in~~	~~10% employee discount~~	covered
	~~New member - logged in~~	**~~5% holiday discount~~**	**covered**
	~~Member - logged in~~	~~None~~	covered
	~~Member - logged in~~	~~10% employee discount~~	covered
	~~Member - logged in~~	~~5% holiday discount~~	covered
3 Login Type-Shipping	~~New member - not logged in~~	~~Standard (5-7 day)~~	covered
	~~New member - not logged in~~	~~Expedited (3-5 day)~~	covered
	~~New member - not logged in~~	~~Overnight~~	covered
	~~New member - logged in~~	~~Standard (5-7 day)~~	covered
	~~New member - logged in~~	~~Expedited (3-5 day)~~	covered
	~~New member - logged in~~	~~Overnight~~	covered
	~~Member - logged in~~	~~Standard (5-7 day)~~	covered
	~~Member - logged in~~	~~Expedited (3-5 day)~~	covered
	~~Member - logged in~~	~~Overnight~~	covered
4 Member Status-Discount	~~Guest~~	~~None~~	covered
	~~Guest~~	~~10% employee discount~~	covered
	~~Guest~~	~~5% holiday discount~~	covered
	~~Member~~	~~None~~	covered
	~~Member~~	~~10% employee discount~~	covered
	~~Member~~	**~~5% holiday discount~~**	**covered**
	~~Employee~~	~~None~~	covered
	~~Employee~~	~~10% employee discount~~	covered
	~~Employee~~	~~5% holiday discount~~	covered
5	~~Guest~~	~~Standard (5-7 day)~~	covered
	~~Guest~~	~~Expedited (3-5 day)~~	covered

Member Status-Shipping	~~Guest~~	~~Overnight~~	covered
	~~Member~~	~~Standard (5-7 day)~~	covered
	~~Member~~	~~Expedited (3-5 day)~~	covered
	~~Member~~	~~Overnight~~	covered
	~~Employee~~	~~Standard (5-7 day)~~	covered
	~~Employee~~	~~Expedited (3-5 day)~~	covered
	~~Employee~~	~~Overnight~~	covered
6 Discount-Shipping	~~None~~	~~Standard (5-7 day)~~	covered
	~~None~~	~~Expedited (3-5 day)~~	covered
	~~None~~	~~Overnight~~	covered
	~~10% employee discount~~	~~Standard (5-7 day)~~	covered
	~~10% employee discount~~	~~Expedited (3-5 day)~~	covered
	~~10% employee discount~~	~~Overnight~~	covered
	~~5% holiday discount~~	~~Standard (5-7 day)~~	covered
	~~5% holiday discount~~	~~Expedited (3-5 day)~~	covered
	~~5% holiday discount~~	~~Overnight~~	covered

Step 5: We get the output in the form of an excel sheet consisting of a test suit with a much lesser number of test cases.

output1 - Microsoft Excel

	A	B	C	D	E	F	G
1	Test Case 1	New member-not logged in	Guest	None	Standard (5-7 day)		
2	Test Case 2	New member-not logged in	Member	10% employee discount	Expedited (3-5 day)		
3	Test Case 3	New member-not logged in	Employee	5% holiday discount	Overnight		
4	Test Case 4	New-member-logged in	Guest	10% employee discount	Overnight		
5	Test Case 5	New-member-logged in	Member	None	Standard (5-7 day)		
6	Test Case 6	New-member-logged in	Employee	None	Expedited (3-5 day)		
7	Test Case 7	Member-logged in	Guest	5% holiday discount	Expedited (3-5 day)		
8	Test Case 8	Member-logged in	Member	None	Overnight		
9	Test Case 9	Member-logged in	Employee	10% employee discount	Standard (5-7 day)		
10	Test Case 10	New-member-logged in	Employee	5% holiday discount	Standard (5-7 day)		
11	Test Case 11	New-member-logged in	Member	5% holiday discount	Standard (5-7 day)		
12							
13							
14							

output1

Ready

Figure 4.8: final output generated from step 5

4.5 Experimental Results

We have applied our algorithm on a number of sets of parameter and there values. Table 4.11 shows various results that have been obtained so far through this algorithm. In this table n_param, n_value, n_test_cases, g_test_cases, reduction represents No' of parameters, number of values to each parameter, total number of test cases, number of test cases generated in a test suit and percentage of test cases reduced, respectively.

Table 4.11: Results Obtained on various set of input

n_param	n_value	n_test_cases	g_test_cases	reduction (%)
3	3	3^3	11	58.2599
3	6	6^3	44	79.6332
3	9	9^3	124	87.6044
4	3	3^4	11	86.4286
4	6	6^4	53	99.9592
4	9	9^4	116	99.9955
5	3	3^5	14	99.4974
5	6	6^5	56	99.9928
5	9	9^5	140	99.7621
6	3	3^6	16	92.5987
6	6	6^6	58	99.8788
6	9	9^6	149	99.9763
8	3	3^8	20	99.6998
8	6	6^8	74	99.9955
8	9	9^8	186	99.9999
10	3	3^{10}	21	99.9996
10	6	6^{10}	87	99.9999
10	9	9^{10}	208	99.9999
20	20	20^{20}	1812	Almost Optimized
20	40	40^{20}	4016	Almost Optimized

4.5.1 Result Analysis

From the above results, following analysis has been made about this algorithm.

- Result obtained from this algorithm proves its efficiency.
- We have received a very less number of test cases in each test suits.

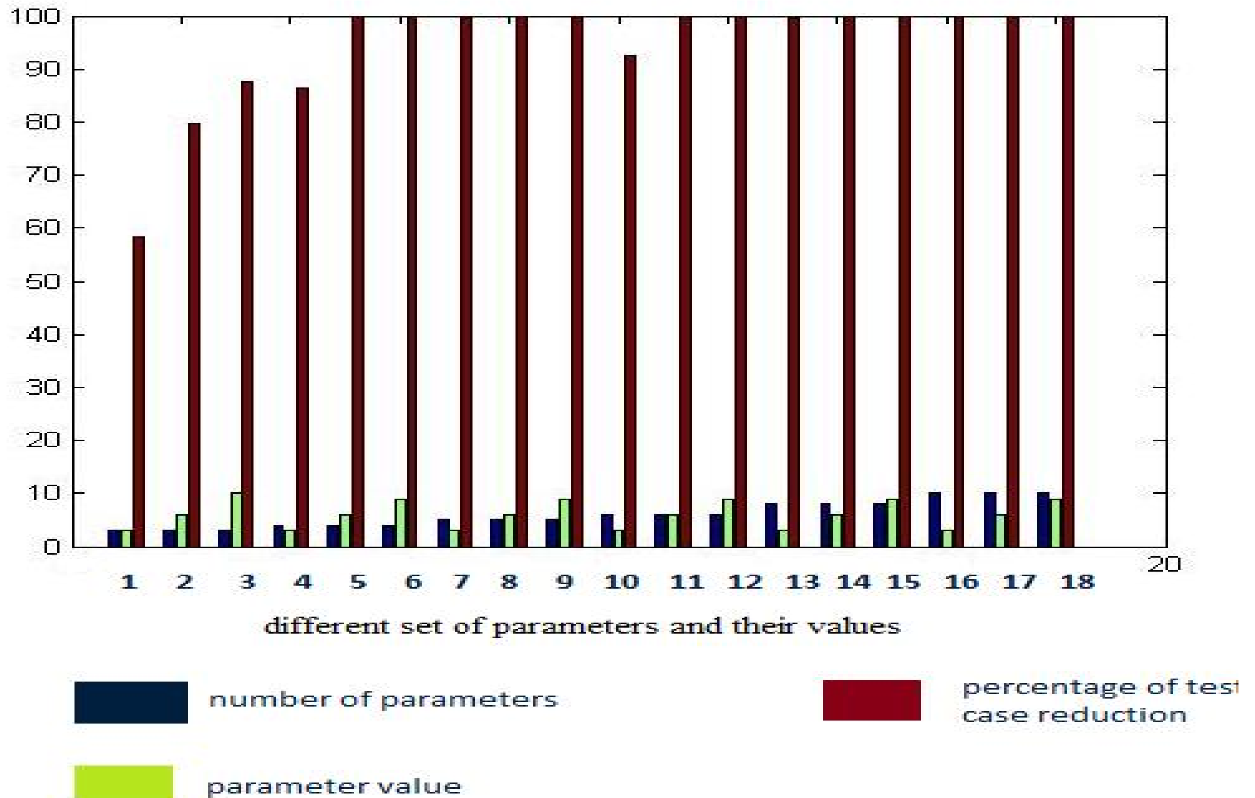


Figure 4.9: Analysis of algorithm

- In figure 4.9, red color bars are showing the percentage of test case reduction. Above figure proves the efficiency of this algorithm as we can seen almost all the cases, we have got the test case reduction beyond 99 percentage.
- We can see observe two trends from the figure 4.9:
    - As the number of value of the parameters increases, keeping number of parameter fixed, then the percentage of test suit reduction increases and efficient results are provided by this algorithm.
    - As the number of parameters increases keeping number of values for those parameters fixed, we get more efficient test suits with lesser number of test cases.

Chapter 5: Conclusion and Future Scope

5.1 Conclusion

Research has proven combinatorial testing as, the best testing method in finding the interaction failures in software applications. To find all combinatorial bugs in a software application, we need to generate test cases. Optimization of test cases is a NP-Complete problem. This thesis proposes an algorithm to get a more efficient and reduced set of test cases to check all the combinatorial bugs. We have studied various techniques for test case optimization and a greedy based approach has been used to generate the test suit with lesser number of test cases.

This algorithm has been implemented in Java language. We have also designed a tool to use this algorithm. Experimental results obtained from this algorithm are satisfactory, giving approximately 99 % of test cases reductions in test suits. Formula generated for quick search in combinatorial testing makes this algorithm fast. Hence, this algorithm reduces the cost and time required checking combinatorial bugs in software applications.

5.2 Future Scope

Following are the future direction in which our proposed work can be carried out:

- We can try to increase the efficiency of this algorithm to generate optimal test suits for computer applications.
- Boundary value analysis and equivalence partitioning can be added in the designed tool.
- For generating better test cases with less efforts, we can combine the features of Artificial Intelligence, and mathematical approach in the proposed algorithm
- There is a scope of integration of combinatorial testing with other types of testing like unit and integration testing.

REFERENCES

1. Borroughs K., and Jain, A., "Improved quality of protocol testing through techniques of experimental design," In Proceedings of the IEEE International Conference on Record, Serving Humanity through Communications, vol. 2, pp. 745–752, 1994.
2. Bryce, R. C., Colbourn, C. J., and Cohen, M. B., "A framework of greedy methods for constructing interaction test suites," In Proceedings of the 27th International Conference on Software Engineering (ICSE'05), ACM, New York, vol. 4, pp. 146–155, 2005.
3. Bryce, R. C. and Colbourn, C. J., "Prioritized interaction testing for pair-wise coverage with seeding and constraints," Inform Software Technol, vol. 5 pp. 960–970, 2006.
4. Bryce, R. C. and Memon, A.M., "Test suite prioritization by interaction coverage," In Workshop on Domain Specific Approaches to Software Test Automation (DOSTA'07). ACM, New York, pp. 1–7, 2007.
5. Bryce, R. C. and Colbourn, C. J., "One-Test-at-a-Time heuristic search for interaction test suites," In Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO'07), ACM, New York, pp. 1082–1089, 2007.
6. Bryce, R. C. and Colbourn, C. J., "A density-based greedy algorithm for higher strength covering arrays," Software Test Verification Reliability, pp. 1–17, 2008.
7. Burr, K., W. Young, "Combinatorial test techniques: Table-Based automation, test generation, and code coverage," International Conference on Software Testing Analysis and Review, pp 503–513, 1998.
8. Cohen M.B., "Designing test suites for software interaction testing," Ph.D. thesis, University of Auckland, New Zealand, 2004.
9. Cohen, D. M., Dalal, S. R., Parelius, J., and Patton, G. C., "The combinatorial design approach to automatic test generation," IEEE Software, pp. 83–88, 1996.
10. Cohen, M. B., Colbourn, C. J., and Ling, A. C. H., "Augmenting simulated annealing to build interaction test suites," In Proceedings of the 14th International

Symposium on Software Reliability Engineering (ISSRE'03), IEEE Computer Society, Los Alamitos, CA, pp. 394, 2003.

11. Cohen, M. B., Dwyer, M. B., and Shi, J., "Exploiting constraint solving history to construct interaction test suites," In Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques (TAICPART-MUTATION'07), IEEE Computer Society, pp. 121–132, 2007.
12. Czerwonka, J., "Pairwise testing in the real world: Practical extensions to test-case scenarios," In Proceedings of the 24th Pacific Northwest Software Quality Conference, 2006.
13. Dalal, S.R, Jain, A., Karunanithi, N., Leaton, J., and Lott, C., "Model-based testing of a highly programmable system," In Proceedings of the 9th International Symposium on Software Reliability Engineering (ISSRE'98), IEEE Computer Society, Los Alamitos, CA, pp. 174, 1998.
14. Dalal, S. R. and Mallows, C. L., "Factor-Covering designs for testing software," Technometrics, pp. 234–243, 2011.
15. Flores P. and Cheon Y., "Generating Test Cases for Pairwise Testing Using Genetic Algorithms," 18th IEEE International Symposium on Software Reliability Engineering (ISSRE'07), pp. 198-200, 2007.
16. Ghazi, S.A. and Ahmed, M., "Pair-wise test coverage using genetic algorithms," In Proceedings of the Congress on Evolutionary Computation (CEC'03), IEEE Computer Society, Los Alamitos, CA, pp. 1420–1424, 2012.
17. Grindal, M., Offutt, J., and Andler, S.F., "Combination testing strategies: a survey Software Testing," Verification and Reliability, vol. 15, issue 3, pp. 167 – 199, 2005.
18. Huller, J., "Reducing time to market with combinatorial design method testing," of the International Council on Systems Engineering (INCOSE) Conference, pp. 176-178, 2000.
19. Kuhn, D.R., Kacker, N., and Lei, Y., "Practical Combinatorial Testing," NIST Special Publication, 2010.
20. Kuhn, D. R. and Reilly, M. J., "An investigation of the applicability of design of experiments to software testing," In Proceedings of the 27th Annual NASA

Goddard Software Engineering Workshop (SEW'02), IEEE Computer Society, Los Alamitos, CA, 91, 2002.

21. Krishna, R. and Nandhan, P. S., "Combinatorial testing: Learning's from our experience," SIGSOFT Software Engineering Notes, 2007.
22. Lott, C., Jain, A., and Dalal, S., "Modeling requirements for combinatorial software testing," SIGSOFT Software Engineering Notes, 2005.
23. Mats, G., Lindstr, B., Offutt, J., and Andler, S. F., "An evaluation of combination strategies for test case selection," Empirical Software Engineering, pp. 583–611, 2006.
24. Mats, G. and Offutt, J., "Input parameter modeling for combination strategies," In Proceedings of the 25th Conference on IASTED International Multi-Conference (SE'07), ACTA Press, pp. 255–260, 2007.
25. Nie C. and Leung H., "A survey of combinatorial testing," 21st International Symposium ACM Computing Society, pp. 11-29, 2011.
26. Mandl, R., "Orthogonal Latin squares: An application of experiment design to compiler testing," Community ACM, pp. 1054–1058, 1985.
27. Renee C.B, Lei, Yu and Kuhn, D.R., DOI:10.4018/978-1-60566-731-7.ch014.
28. Sherwood, G., 1994. "Effective testing of factor combinations, " In Proceedings of the 3rd International Conference on Software Testing, Analysis, and Review (STAR94), pp. 435-437, 1994.
29. Shiba, T., Tsuchiya, T., and Kikuno, T., "Using artificial life techniques to generate test cases for combinatorial testing, " In Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04), IEEE Computer Society, Los Alamitos, CA, pp. 72–77, 2004.
30. Sampath, S. Ryce, R., Viswanath, G., Kandimalla, V., and Koru, A., "Prioritizing user-session-based test cases for web applications testing, " In Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, pp. 141–150, 2008.
31. Salem, A., Rekab, M. K., and Whittaker, J. A., "Prediction of software failures through logistic regression," Inform. Software. Technol, pp. 781–789, 2004.

32. Schroeder, P. J., “Black-Box test reduction using input-output analysis,” Ph.D. thesis, Illinois Institute of Technology, Chicago, 2002.
33. Schroeder, P. J., and Korel, B., “Black-box test reduction using input-output analysis,” SIGSOFT Software. Engineering. Notes, 2000.
34. Schroeder, P. J. and Korel, B., “Constraint of model-based test generation using input-output analysis,” In Proceedings of the Automated Software Engineering Doctoral Symposium. ACM, pp. 35–43, 2000.
35. Turban, R. C., “Algorithms for covering arrays,” Ph.D. thesis, University of Ahzam Tempe, Ukraine, 2006.
36. Tung, Y. W. and Aldiwan, W., “Automating test case generation for the new generation mission software system,” In Proceedings of the IEEE Aerospace Conference. pp. 431–437, 2010.
37. White, L., and Almezene, H., “Generating test cases for GUI responsibilities using complete interaction sequences,” 11th International Symposium on Software Reliability Engineering (IISRE), IEEE Computer Society, pp. 110-121, 2000.
38. Williams, A.W. “Determination of test configurations for pair-wise interaction coverage,” In Proceedings of the 113th International Conference on Testing Communicating Systems (TestCom’00), Kluwer, pp. 59–74, 2000.
39. Williams, A. W., “Software component interaction testing: coverage measurement and generation of configurations,” Ph.D. thesis, University of Ottawa, Canada, 2002.
40. Williams, A. and Probert, R.L., “A practical strategy for testing pair-wise coverage of network interfaces,” 7th International Symposium on Software Reliability Engineering (ISSRE’96), IEEE Computer Society, pp. 246, 1996.
41. Williams, A. W. and Probert, R. L., “A measure for component interaction test coverage,” In Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications (AICCSA’01). IEEE Computer Society, Los Alamitos, CA, pp. 304, 2001.

PUBLISHED

1. Chooramani, A., and Garhwal, S., "Combinatorial Testing: A Systematic Review," International Journal of Advance Research in Computer Science and Software Engineering (IJARCSSE), Vol. 3, Issue. 6, July, 2013.

ACCPEPTED

1. Chooramani, A., and Garhwal, S., "A Greedy Based Approach for Generating Minimal Covering Array and Optimal Test Suit for Combinatorial Testing," International Journal Computer Application (IJCA) Vol. 75, August, 2013.