# APPLYING COMBINATORIAL TESTING TO SYSTEMS WITH A COMPLEX INPUT SPACE

by

MEHRA NOUROZ BORAZJANY

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2013

ACKNOWLEDGEMENTS

Over the past four years I have received support and encouragement from a great number of individuals. Dr. Y. Lei has been a mentor, colleague, and friend. His guidance has made this a thoughtful and rewarding journey. This thesis would not have been possible without the help, support and patience of him. I would like to thank my committee of Dr. Kung, Dr. Liu and Dr. Csallner for their support as I moved from an idea to a completed study. In addition Dr. Khalili provided the needed encouragement and support.

I would also like to thank all who took part in this study for generously sharing their time and ideas. I have learned much through our conversations. I would like to thank my fellow doctoral students—those who have moved on, those in the proceeding, and those just beginning—for their support, feedback, and friendship. I would like to thank my friends for accepting nothing less than completion from me. I would like to thank my family for their support - my parents for believing in me, my sisters for their advice, and my husband for his inspiration and great patience at all times. Last, but certainly not least, I would like to thank the faculty and staff at the CSE department for their strength and dedication to the profession.

April 26, 2013

## ABSTRACT

APPLYING COMBINATORIAL TESTING TO SYSTEMS

WITH A COMPLEX INPUT SPACE

MEHRA NOUROZ BORAZJANY, PhD

The University of Texas at Arlington, 2013

Supervising Professor: Yu Lei

Combinatorial testing, which has been shown very effective in fault detection, is a testing strategy that applies the theory of combinatorial design to test software programs. Given a program under test with k parameters, t-way combinatorial testing requires all combinations of values of t (out of k) parameters be covered at least once, where t is usually a small integer. Combinatorial testing can significantly reduce the cost of testing while increasing its effectiveness.

Input space modeling is an important step in combinatorial testing. The input space of a subject program must be modeled before combinatorial testing can be applied to it. The effectiveness of combinatorial testing to a large extent depends on the quality of the input space model. If the input space is modeled perfectly, all faults caused by interactions involving no more than t parameters will be detected.

In this dissertation, we develop an input space modeling methodology for combinatorial testing. The main idea is to consider the process of input space modeling as two steps, including input structure modeling and input parameter modeling. The first step tries to capture the structural relationship among different components in the input space. The second step tries to identify parameters, values, relations and constraints for individual components.

We present several case studies of applying the proposed methodology to five real-life programs. These studies are designed to validate the proposed methodology in a practical setting. They are also designed to evaluate the effectiveness of combinatorial testing applied to real-life programs. We compare the proposed methodology to two random approaches: (1) pure-random, which generates test sets with minimum effort on modeling; and (2) modeled-random, which generates random tests from the same model created by the proposed methodology. The results show that proposed approach is more effective than the modeled-random approach, which is significantly more effective than the pure-random approach.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

# CHAPTER 1

# INTRODUCTION

There are two general software testing strategies including white-box testing and black-box testing. In white-box testing, test inputs are derived from the implementation. In black-box testing, test inputs are identified from the specification. Combinatorial testing is a black-box testing technique. As a result, combinatorial testing requires no knowledge about the implementation of the system under test. The specification required by combinatorial testing is typically lightweight, as it only needs to identify parameters and values, as well as relations and constraints that may exist between these parameters, which are taken by the system under test.

Software systems tend to have more parameters as they become larger or complex or both. Interactions of these parameters may cause failures. As a result, software testing often needs to test a large number of combinations among different parameters. Exhaustive testing, i.e., testing all the parameter value combinations, is generally infeasible. Even if we have the resources to test all combinations, this may not be effective because many combinations do not actually cause any failure. Thus, there is a need to develop an effective strategy that selects a subset of these combinations to be tested. Combinatorial testing is developed as one such strategy.

Combinatorial testing is a testing strategy that applies the theory of combinatorial design to test software systems. T-way combinatorial testing of a system with $k$ parameters covers all the value combinations of $t$ (out of $k$) parameters at least once, where $t$ is usually a small integer no more than six [8]. Software failures are often the result of a faulty interaction between parameters. If parameters are modeled perfectly, all the faults caused by interactions involving no more than $t$ parameters will be exposed by t-way combinatorial testing. Combinatorial testing can significantly reduce the cost of software testing at the same time as increase its effectiveness [31][32].

Combinatorial testing can be applied to a system only after the input space of the system is modeled. An input space model contains a set of parameters, each of which has a set of possible values, and possible relations and constraints between these parameter values [4]. The effectiveness of combinatorial testing depends to a large extent on the quality of the input space model. In particular, if a failure can only be triggered when a parameter takes a specific value, and if this parameter or value is not modeled, this failure will not be detected by a combinatorial test set. Thus, input space modeling is a critical step in the process of combinatorial testing. There are important design decisions and tradeoffs to be made in the modeling process. Different testers may come up with different models, depending on their creative choices and experiences [1].

A number of studies have been reported on input space modeling for general software testing, i.e., not specific to combinatorial testing. Grochtmann and Grimm [3] mentioned that finding parameters and values is a creative process that can never be fully automated. Several methods could be used for input parameter modeling, such as Category Partition [4] or Classification Trees [3]. Both of the methods try to create a model of the input domain. The Category Partition method partitions the input domain into categories and choices. The Classification Tree method partitions the input domain into classifications and classes. The classifications are like categories and classes are the choices. The Classification Tree method improves the Category Partition method. The Classification Tree graphically represents a partition of the input domain in the form of a tree. The main difference is how the constraints among classes are obtained. While the Category Partition obtains the constraints as a list of categories, choices, and constraints in textural format, the Classification Tree obtains these constraints as a tree structure.

Only a few studies have been reported on input space modeling for combinatorial testing. A workflow of eight steps is proposed for the combinatorial modeling process [1]. In the summary section the following steps suggested by this process: selecting a modeling approach either interface-based or functionality-bases, identifying parameters, values, constraints, and translating the test cases to executable test cases.

Segall et al. suggests several patterns that commonly appear in the input space modeling process for combinatorial testing such as optional values and multiplicity [28][29]. The optional values are useful to distinguish between the cases of a valid value and an empty parameter. The multiplicity pattern is used when multiple parameters of the same type appear within a system. They encountered these patterns in many different models, regardless of the domain of the system under test or the current level of the testing. Therefore, they described these patterns along with simple and effective solutions for them.

This dissertation proposes an input space modeling strategy for combinatorial testing. We consider the process of input space modeling as two steps: input structure modeling (ISM) and input parameter modeling (IPM). The first step, i.e., ISM, tries to capture the structural relationship among the different components in the input space. The second step, i.e., IPM, tries to identify parameters, values, relations and constraints for individual components. We also suggest strategies about how to perform unit and integration testing based on the input space structure.

The focus of this dissertation is mainly on the first step. Existing methods such as category partitioning can be used for the second step. We consider two types of structures, i.e., flat and graph. The flat structure has no compositional hierarchy, where components are equal peers in terms of composition relation. For example, a flat input structure can be used to model the command-line options of a program. The graph structure represents the composition relation between different components in a graph, where one component may be composed of several other components. For example, a graph structure can be used to model elements in an XML file.

We report case studies of applying the proposed methodology to five real-life programs, including three programs from the SIR website [20], i.e., *Apache Ant*, *space*, *make*, and one program from GNU [18], i.e., *grep*, and a combinatorial test generation tool called *ACTS* [12]. ACTS is developed jointly by the US National Institute of Standards and Technology and the University of Texas at Arlington, and currently has more than 1200 individual and corporate users. This subject was conceived when a user of ACTS asked the question: Have you tested ACTS using ACTS?

The subject programs are selected because of several desired attributes, including their complex input space, the existence of a clean version and multiple faulty versions, a relatively large number of lines of code, and the availability of their specifications. *Apache Ant* and *ACTS* which are written in java, contain 80500 and 24637 line of code respectively and *Space*, *Make, and Grep* which are written in C, contain 9127, 35545, and 10068 line of code respectively.

The case studies are designed to serve two purposes. First, they are designed to validate the proposed methodology in a practical setting. Second, they are intended to evaluate the effectiveness of combinatorial testing. There has been a lack of empirical studies and experience reports on applying combinatorial testing to real-life programs [9].

In our case studies, we compare combinatorial testing based on the proposed methodology to two random approaches. The first random approach, referred to as pure-random, generates random test sets with minimum effort on modeling, i.e., mainly based on the syntactic structure of the input space. The second random approach, referred to as modeled-random, generates random test sets from the same model created by the proposed methodology. We measure the effectiveness of these approaches in terms of code coverage and number of faults they detect. The results show that our approach is more effective than the modeled-random approach, which is significantly more effective than the pure-random approach. The implication of the results is two-fold. First, input space modeling plays an important role in determining the effectiveness of a testing process. Second, combinatorial testing as a test generation strategy can be more effective than random testing.

The goal of this thesis is to develop a systematic methodology for modeling complex input space for combinatorial testing. Through this methodology, the effectiveness and popularity of combinatorial testing can be improved. In Chapter 2, we discuss related work on combinatorial testing in general. In addition, we discuss the major works related to an input parameter modeling. Furthermore, we briefly discuss the previous empirical studies on combinatorial testing by other researchers. In Chapter 3, we describe our methodology for input space modeling. It consists of three parts. The first part describes the input structure modeling along with suggested testing methods. The second part describes the input parameter modeling. The third part explains the concrete test case generation from abstract test cases. Chapter 4 reports and discusses our case studies. It consists of six parts. The first part focuses on subject programs. The second part gives a brief overview of input models for each subject. The third part explains the process of test generation. The forth part discusses the metrics. The results per subject represents in the fifth part. Here we also explain the step by step process of modeling for each subject. The last part of Chapter 4 summarizes the results of these studies. These studies aim to investigate (1) Whether the proposed methodology is valid in a practical setting; (2) The effectiveness of combinatorial testing in terms of fault detection and coverage; (3) The importance of the modeling process for combinatorial testing. We compared combinatorial testing based on the proposed methodology to two random approaches. The first random approach, referred to as pure-random and the second random approach, referred to as modeled-random. Finally, Chapter 5 concludes this dissertation by summarizing the contribution of our work, and suggests possible directions of future work.

# CHAPTER 2

# RELATED WORK

## 2.1 Overview

Software failures are often the result of a faulty interaction between input parameters. Testing all the combinations of the input parameters, i.e. exhaustive testing, is often impossible for large and/or complex software systems due to resource constraints.

Combinatorial (or t-way) testing covers every combination of any t parameter values at least once [40]. Empirical studies suggest that combinatorial testing can be very effective for fault detection in practice. In particular, a NIST study suggests that all the faults in several applications are caused by interactions among six or fewer parameters [8].

Before combinatorial testing can be applied to any system, the input space of the system should be modeled. The quality of the input space model has a significant impact on the effectiveness of combinatorial testing. Imagine a situation that a tester forgot to model a parameter; as a result, the combinatorial test set will not be able to detect the fault that triggered by this parameter.

This dissertation proposes an input space modeling strategy for combinatorial testing. We consider the process of input space modeling as two major steps: input structure modeling (ISM) and input parameter modeling (IPM). The remainder of this chapter is organized as follows. Section 2.2 discusses related work on combinatorial testing in general. Section 2.3 discusses related work on input parameter modeling. Section 2.4 discusses related work on empirical studies in combinatorial testing.

## 2.2 Combinatorial testing

Combinatorial testing, which has proven very effective in fault detection, is a testing strategy that applies the theory of combinatorial design to test software systems. Given a system under test with k parameters, t-way combinatorial testing requires all combinations of values of t (out of k) parameters be covered at least once, where t is usually a small integer. If test parameters are modeled perfectly, all faults caused by interactions involving no more than t parameters will be detected. Combinatorial testing can significantly reduce the cost of software testing while increasing its effectiveness.

Many combinatorial test generation strategies have been proposed to generate test sets that are as small as possible but still satisfy t-way coverage. Grindal et al. [44] surveyed fifteen important strategies that have been reported in the literature. Two representative strategies, i.e., the AETG strategy [40] and the IPO strategy [92], are described as follows.

The AETG (Automatic Efficient Test Generation) strategy adopts a greedy framework for combinatorial test generation. In this algorithm, a test is created to cover as many uncovered t-way combinations as possible. First, it selects a value of a parameter that appears in the most uncovered combinations. Second, it randomly selects another parameter from the rest of the parameters to cover the most number of uncovered pairs. Third, it includes values of the remaining parameters one by one, with the policy used at the second step. Fourth, it repeats the above steps to generate a certain number of candidate tests, and picks the candidate test that covers the most uncovered t-way combinations as the final test. It repeats these steps until all t-way combinations have been covered.

Lei et al. [92] proposed another t-way testing strategy called In-Parameter-Order (IPO). The IPO strategy generates a t-way test set to cover all the t-way combinations between the first t parameters and then extends the test set to cover all the t-way combinations of the first t+1 parameters. This process is repeated until the test set covers all the t-way combinations of the parameters. In this dissertation, we used a tool called ACTS to generate our test cases. ACTS implements the IPO strategy.

The following examples show 2-way test sets generated using the mentioned algorithms for a system contains three parameters P1, P2 each have two values [A,B] and P3 that have three values [A,B,C].

Table 2.1 AETG pair-wise test cases

| P1 | P2 | P3 |
|---|---|---|
| **A** | **A** | **A** |

First test

| P1 | P2 | P3 |
|---|---|---|
| A | A | A |
| **B** | **B** | **A** |

Second test

.....

| P1 | P2 | P3 |
|---|---|---|
| A | A | A |
| B | B | A |
| A | B | B |
| B | A | B |
| A | A | C |
| **B** | **B** | **C** |

Last test

Table 2.2 IPO pair-wise test cases

| P1 | P2 |
|---|---|
| A | B |
| B | A |
| A | A |
| B | B |

| P1 | P2 | **P3** |
|---|---|---|
| A | B | **A** |
| B | A | **A** |
| A | A | **B** |
| B | B | **B** |

Horizontal Growth

| P1 | P2 | P3 |
|---|---|---|
| A | B | A |
| B | A | A |
| A | A | B |
| B | B | B |
| **A** | **A** | **C** |
| **B** | **B** | **C** |

Vertical Growth

## 2.3 <u>Input parameter modeling</u>

A number of studies have been conducted on input parameter modeling for general software testing, i.e., not specific to combinatorial testing. Grochtmann and Grimm [3] mentioned that finding parameters and values is a creative process that can never be fully automated. Several approaches, e.g., Category Partitioning [4] and Classification Tree [3] have been reported for input space modeling for general software testing. The Classification Tree method step by step divides the input domain into classifications and classes. The partition of the input domain into classifications is represented graphically in the form of a tree. The Category Partition method divides the input domain into categories and choices The Classification Tree method improves the Category Partition method. The main difference is the constraints representation among classes. The Classification Tree represents constraints among classification and classes as a tree structure, while the Category Partition represents the constraints in textural format. Although tree-based description can capture some dependencies in the test space but it does not address more complex relationships between the different elements of the test space. For example auxiliary aggregate or commonality discussed in [29] is used to in order to reduce the number and the complexity of the restrictions. We will explain it in the following paragraphs.

Chen et al. [2] presented a list of common mistakes in identification of the category (parameter) and choices (values) from specification. The missing categories, problematic categories, and problematic choices are the main mistakes. They introduced a checklist of six steps for detecting these mistakes:

1. Any irrelevant categories identified for factors that are not related to the execution of the selected unit under test should dismiss.
2. If a factor is not related to any identified category then a category is missed.
3. If the set of complete test frames for a potential choice in a category is empty then this choice is invalid.
4. If the union of all identified valid choices does not cover the input space of that category then a choice is missed.
5. There should be no overlapping between the valid choices of each category.
6. The constraints among choices should be considered.

For example a choice is a 'problematic choice' if it is an invalid choice or one of the overlapping choices. Moreover, a category is a 'problematic category' if it is an irrelevant category or a category with a problematic/missing choice. The above checklist cannot guarantee to detect all the mistakes. However, the empirical studies reported in this paper suggest that it can greatly reduce such unwarranted cases.

Only a few studies have been reported on modeling for combinatorial testing. A workflow of eight steps is proposed for the combinatorial modeling process [1]. The first step suggested by this process is selecting a modeling approach either interface-based or functionality-bases. In addition, the paper suggested applying functionality-based approach due to a number of reasons. For example requirements are often available before the actual implementation. During steps two to five, parameters, values, constraints would be identified.

The final steps focus on translating the test cases to executable test cases. The focuses of this study was mainly on the workflow of the modeling process, i.e., not on the actual modeling process.

Segall et al. [28][29] and Lott et al. [27] studied some patterns that commonly occur in combinatorial test models. Common patterns include optional values, multiplicity, and auxiliary aggregates or commonality. The optional values are used when having a value for a specific parameter is optional based on the specification. Therefore, to distinguish between the cases of a valid value and an empty parameter, the optional values such as 'not applicable' or 'empty' will be used. The multiplicity pattern is used when multiple parameters of the same type appear within a system. For example if our system has two parameters p1 and p2 which they have their own values, for example p1 has values v11, v12 and p2 has values v21, v22, and we can use these parameters multiple times, the multiplicity pattern suggests to model such a system with four parameters with three values [zero, one, more]. This will help to avoid redundancy in the model and in the resulting test plan. The concept of auxiliary aggregate is to factor out the common parameters with identical concrete values in the model. In contrast, we reuse the abstract model and the concrete values can be different. Therefore, the concept of reusing the commonality or auxiliary aggregate is different from our approach as discussed later in chapter 3.3. Also they reported three categories of pitfalls in combinatorial models: correctness, completeness, and redundancy. A model that does not capture correctly what it is intended to capture is an incorrect model. A model that omits an important part of the test space is an incomplete model and a model that explicitly enumerates different cases that are actually equivalent is a redundant model. The optional value pattern falls into the correctness category and multiplicity fall into the redundancy category.

In [10], our graph structure is referred to as sub-attributes and it was suggested to either consider the parent node as a compound parameter or split the parent node into simple parameters. In their experiments they used the split approach because it was believed that this approach would generate a less number of test cases. The split approach effectively converts the graph structure into a flat structure, which creates more parameters and may also introduce many invalid combinations. The split approach may also introduce redundant factors if one child has more than one parent.

To complement mixed-strength test generation, a user is allowed to create a hierarchy of test parameters [43]. For example if our system has five parameters and one of the parameters is a compound parameter with three values, then we can test the compound parameter with a different strength than the other five parameters. This is similar to our graph structure without loop as it is shown in Figure 3.2. However, the hierarchy in [43] can have two levels only while our approach does not have this restriction. Moreover our approach deals with loops in a graph structure.

The above works are complementary to our work. None of the above addresses the problem of input structure modeling.

## 2.4 Empirical studies in combinatorial testing

Several empirical studies of combinatorial testing have been reported that applied combinatorial testing to various types of applications. In [8], Kuhn et al. reported a study of several fault databases and found that all the faults in these databases are caused by no more than six factors. They analyzed 329 error reports of a large system with a number of subsystems in NASA. Different systems such as database, server, and browser with various sizes (LOC) from 3000 to $2\times10^6$ are used in this study. Faults are characterized in a database by date submitted, severity, priority for fix, the location where found, status, the activity being performed when found, and several other features. The summary of all failures reviewed in this paper were triggered by no more than six factors. This study suggested that if all errors in a particular class of software are triggered by finite combinations of t parameters or less, then testing all combinations of t or fewer parameters would provide a form of pseudo-exhaustive testing; therefore, there is no need to perform exhaustive testing on that particular class of software.

Combinatorial testing was applied to a mobile phone program [10]. This work mainly focuses on illustration of the steps involved in applying combinatorial testing, and did not report the actual testing results. They reported on how to test an in-house web-based application that they designed using a customized version of OATS (Orthogonal Array Based Testing Strategy). They shared details on using this application in feature testing of a mobile phone application. First, they partitioned the requirements into five groups such as continue/end task functionality, slider tone functionality and so on. In the second step they identified their scope of testing e.g. UI testing or functionality testing. Third, they identified the variables. It was suggested that every noun in the requirement could be a possible variable. In the fourth step they tried to assign levels to the parameters. They suggested that for example parameter Head Set and Speaker Phone should merge together. This is because; speaker phone functionality will not be available when the head set is connected. If the level of a parameter is fully dependent on the level of the other parameter, they should be merged into a single parameter. In the fifth step, they identified the constraints and finally generated the test cases by using the OATS tool. The final model for the mobile program consists of six parameters with a maximum of nine values for each parameter and 10 constraints.

The results of applying combinatorial testing to a real-life email system were reported in [11]. They used the AETG tool to generate pairwise test cases and code coverage is then used to indicate missing functionality. They applied the mentioned method on Nortel's internal e-mail system where they were able to cover 97% of branches with less than 100 valid and invalid test cases, as opposed to 27 trillion exhaustive test cases. However, this work did not report the details of the input space model, e.g. number of parameters, values and constraints. The main source used to identify parameters and values was RFC822 (Standard for the Format of Arpa Internet Text Messages), since specification was not available.

A study that applied combinatorial testing to browser compatibility has been reported [13]. The main idea is to offer two methods, i.e. single factor coverage and pair-wise coverage, to obtain the suites of compatibility test cases. The single factor coverage changed the value of one factor once at a time and kept the typical values of other factors. They also suggested that if a system has many parameters values that every value interferes with others this method is not practical. The pair-wise coverage covered 2-way combinations between parameters; therefore, it has better quality than single factor coverage. This paper however did not report any testing results.

Combinatorial testing has also been applied to the domain of protocol testing [15]. The main idea is to show how the quality and efficiency of protocol testing could be improved by using combinatorial testing compared to traditional approaches. However, this work did not explain the details of the modeling process. They discussed two examples that illustrate the application of the AETG tool to protocol conformance testing: Call Rejection and Channel Negotiation. They also summarized the number of test cases needed and the breadth of coverage for the two traditional approaches and the AETG approach. The first traditional approach includes a complete coverage of the test space, but it requires a large number of test cases. The second traditional approach significantly reduced the number of test cases, but it suffered from a lower coverage. The AETG approach provides a much broader coverage of the test space with a minimum number of test cases. For the call rejection system six parameters with a maximum of seven values for a parameter are identified and the number of test cases/code coverage for each testing approach is 504/100, 46/33, 42/100 respectively. Similarly, for the channel negotiation system four parameters with a maximum of six values for a parameter are identified and the number of test cases/code coverage for each testing approach are 108/100, 21/30, 18/100 respectively.

Wang et al. [6] presented a test sequence generation approach for covering all interactions between any two pages of a web application. The empirical results of applying their approach to five open source applications show that the approach significantly reduces the number of submission tests that have to be performed while still achieving a high degree of coverage of dynamic pages. The assumption was that values of individual parameters are supplied by using other techniques or generated manually by the user. Using pair-wise coverage can significantly reduce the number of requests that have to be submitted while still achieving effective coverage of the navigation structure. We used this technique in combinatorial testing of the graphical user interface (GUI) of the ACTS tools. We covered all interactions between any two pages of the GUI.

Lei et al. applied combinatorial testing to concurrent programs [35]. This strategy is based on an existing technique called reachability testing which is exhaustive. The main idea is to cover all the t-way combinations, instead of all the combinations, of the changes that can be made to the race outcomes in a test sequence. The empirical studies of five programs indicate that the new strategy can substantially reduce the number of test sequences that need to be exercised while still effectively detecting faults. The t-way reachability testing considers each receive event as a parameter, and each possible race outcome change that can be made to a receive event as a value of the parameter representing the receive event.

Lammel et al.[99] presented a grammar-based testing technique using controllable combinatorial coverage. The idea is to get the full combinatorial coverage of a grammar up to certain depth. For example, we can limit the recursive applications for a ‘nonterminal’ and decrease its ‘productivity’. They also implemented in a tool, Geno. For example the goal is to generate test-data for testing the following example.

```
Exp = BinExp ( Exp , BOp, Exp ) // Binary expressions
| UnaExp ( UOp , Exp )          // Unary expressions
| LitExp ( Int ) ;              // Literals as expressions
```

They also assumed that they have access to a test-oracle; so we only care about test-data generation at this point. They executed the grammar with Geno to generate all terms in order to increasing depth. Then the number of terms with increasing depth is calculated. For the above example, there is no term of sort Exp with depth 1, there is 1 term of sort Exp with depth 2: LitExp("1"); and 2 terms of sort Exp with depth 3 and 3349 terms of sort Exp with depth 6. They called it combinatorial complexity of this grammar. A full combinatorial coverage for a grammar which has nonterminals, declarations, statements and expressions that are used in statement contexts is impractical. Therefore, they tried to find an approximate combinatorial coverage. For the above example they required one-way testing for the constructor of binary expressions. Also they limited the recursive depth of expressions used in the construction of unary expressions called as MaxRecDepth. The one-way and MaxRecDepth approximations reduced the number of terms of sort Exp with depth 6 to 45 and 651 respectively. The main idea of the approach was that test data is generated in a combinatorially exhaustive manner and approximations defined by the test engineer. Test engineers can generate test cases that focus on particular problematic areas in language implementations like capacity tests, or the interplay between loading, security permissions and accessibility.

Wang et al. [33] presented a buffer overflow detection approach. It adapts combinatorial testing to the domain of security testing. The idea is that the attacker can change the behavior of a target system by controlling the values of its external parameters e.g. input parameters, configuration options, and environment variables. The attacker intends to change the critical data structure e.g., a return address on the call stack such that he or she is allowed to gain control over the program execution. The critical data is located close to a buffer that is vulnerable to overflow. An attack-payload parameter is an external parameter that is often chosen such that during program execution, it is possible that its value be copied into this buffer. It is proven by this study that in many buffer-overflow attacks a single attack-payload parameter exists. An external parameter is identified to be an attack-control parameter of an attack-payload if its value affects the value of attack-payload. Their approach mainly focused on how to generate tests such that the two following conditions are satisfied. First, during the execution of a test, the statement must be executed. Second, during the execution of the same test, buffer must be overrun. This happens when a variable is excessively long, or the buffer is excessively small, or both. In their approach, attack-payload and attack-control parameters and their values are identified manually based on specification or domain knowledge or both. They also provided guidelines on how to identify attack-payload and attack-control parameters and a set of values for each of these parameters. For each integer attack-payload parameter, they identified three extreme values [positive number, zero, small negative]. For each string parameters, they identified a single extreme value that is typically longer than normal. For each attack-payload parameter they identified an attack-control parameter only if a connection existed between them. They reported the number of parameters, number of attack-payload parameters, average number of values per attack-payload parameter, average number of attack-control parameters per attack-payload parameter, average number of control values per attack-control parameter for each program. They also reported the number of test cases that they generated for each program. For each extreme value of each attack-payload parameter they generate a group of tests. They implemented a tool called Tance and conducted experiments on five open-source c programs: Ghttpd(609LOC), Gzip(5809LOC), Hypermail(23057LOC), Nullhttpd(2245LOC), and Pin(154301LOC). They examined vulnerability reports in three public vulnerability databases. The results showed that their approach detected all the known vulnerabilities but one for the first four programs.

Kuhn et al. [22] compared the effectiveness of random and t-way combinatorial testing for deadlock detection on a grid computer network simulator with tests covering 2-way to 4-way combinations of configuration values, paired with an equal number of randomly generated tests. The software under test for the experiment was Simured, a multicomputer network simulator consisting of 2,131 lines of C++ code. No seeded faults or other modifications were made to the Simured software. A total number of 14 parameters with maximum 5 values were identified. While our comparisons between combinatorial and random testing focuses on fault detection, this study evaluates these methods with respect to deadlock detection in a simulation. They used deadlocks as events of interest to make evaluating program responses straightforward and unambiguous. They did not use numerical results such as packet rates or delays. In addition the software under test is a small but complex program that is not assumed to have characteristics similar to our subject program. For example the network simulation requires extensive statistical calculations such as packet transmission rates and delays, and is not directly comparable to our subject programs. The study is complementary to our work in that they all provide evidence and insights on the effectiveness of combinatorial testing in practice. However, they did not report the details of the modeling process.

Schroeder et al. [24] designed a controlled study to compare the fault detection effectiveness of n-way and random test suites. Combinatorial testing is conducted on two subject programs, Data Management Analysis System (8.7k LOC) and Loan Arranger System (6.2k LOC), that have been injected with software faults. For the first subject they tested only the product sample concentration report, which requires the tester to consider combinations of 18 different input variables. For the second subject they tested only the system's administrative mode that is used to update and maintain the system's data repository, which requires the tester to consider combinations of 19 different input variables. They stated that these systems require a significant amount of combinatorial testing effort. They did not attempt to inject complex faults. The faults that are likely detected by all test cases are removed. For the first subject they injected 82 faults and for the second subject they injected 88 faults. To produce t-way and random test suites they used a tool called the Test Vector Generator (TVG) which implements a greedy algorithm for combinatorial test generation. The TVG tool also generates random combinatorial test suites of any size. They executed the test suites using a data-driven test automation platform which allows automated execution of the test cases and evaluates the result of each test case by comparing its result to the result produced by the original copy of the system. They used equivalence partitioning and boundary value analysis to select the test data values. The author having the most domain knowledge about the applications selected the test data values of the experiment. The test data values selected reflect positive test cases only. This study finds no significant difference between the two methods. Although they did not explain how they actually model the system but the number of parameters that they identified for such large systems with more than 6k LOC is very small compare to our study. In addition if modeling is required then random testing will lose its advantage over combinatorial testing.

Qu et al. [36] examines the effectiveness of combinatorial testing for regression testing across multiple versions of two subjects, flex and make from SIR website with provided seeded faults. Also they seeded additional 30 faults into each subject. They generated t-way covering arrays using simulated annealing with t set to 2 and 3 for flex, and 2, 3 and 4, and 5 for make. They also generated random array of the same size for each subject but they did not mentioned how they generated the random arrays. Also they did not report how they modeled the input space but they reported the number of parameter values. For flex their model contains 7 parameters with a maximum of 16 values for a parameter and for make their model contains 8 parameters with a maximum of 5 values for a parameter. The result of this study indicated that as the strength of t was increased the fault detection improved. Also the t-way test set had a better fault detection than the random test set of the same size. In addition, their experimental results suggested that the combinatorial testing is an effective method of testing to be used in regression testing.

The random testing approach used these studies are the modeled-random approach in our study. Our study does not only compare to the modeled-random approach but also the pure-random approach.

Dalal et al. [37] conducted an investigation to find out how many 2-way combinations covered by a test set generated by random testing. Their results show that with the same number of test cases as 2-way testing, random testing covers more than 90% of 2-way combinations.

We also conducted similar investigations to find out how many t-way combinations are covered by a randomly generated test set. The results show that with the same number of test cases as t-way testing, modeled-random covers more than 90% of (t-1)-way combinations and 81% to 87% of t-way combinations and 45% to 55% of (t+1)-way combinations.

# CHAPTER 3

# AN INPUT SPACE MODELING METHODOLOGY

Before we can apply combinatorial testing to a system, the input space must be modeled. A low-quality input space model can reduce the effectiveness of combinatorial testing. A failure will not be detected by a combinatorial test set if the parameter or value that triggers this failure is not modeled.

For a system that has a large number of features, we first use a divide-and-conquer strategy to divide the system into smaller systems. There are two general strategies. One is to divide the system vertically, e.g., based on features. That is, we could apply combinatorial testing to one feature or a group of related features at a time. The other strategy is to divide the entire system into several subsystems, where each subsystem may be involved in multiple features.

Next, we model the input space of each module. This modeling process consists of two major steps, Input Structure Modeling (ISM) and Input Parameter Modeling (IPM). ISM tries to capture the structural relationship among the different components in the input space. We consider two types of structures, i.e., flat and graph. The flat structure has no hierarchy, i.e., components are equal peers. For example, a flat input structure can be used to model the command-line options of a program. The graph structure represents the composition relation between different components in a graph, where one component may be composed of several other components. For example, a graph structure can be used to model elements in an XML file.

In [10], the graph structure is referred to as sub-attributes and it was suggested to either consider the parent node as a compound parameter or split the parent node into simple parameters. In their experiments they used the split approach because it was believed that this approach would generate a less number of test cases. The split approach effectively converts the graph structure into a flat structure, which creates more parameters and may also introduce many invalid combinations. The split approach may also introduce redundant factors if one child has more than one parent. In this study we consider the first approach where our graph structure represents the composition relation between different components. This will help to reduce the complexity of input space modeling.

IPM tries to identify parameters, values, relations and constraints for individual components. Existing IPM methods can be applied. In particular, it is often necessary to identify abstract parameter and values. One common approach is to identify factors that could affect the behavior of the object being modeled. Each of these factors can be identified as an abstract parameter. Then, existing methods such as category partitioning and classification tree can be used to identify the abstract values of each abstract parameter.

After we have the input parameter model for each module, we generate test cases from the model using combinatorial testing tools such as ACTS. These test cases are abstract test cases because the parameters and values in the model are abstract. Thus, it is necessary to derive concrete test cases from these abstract test cases before the actual testing can be performed. Note that an abstract test case typically represents a set of concrete test cases, from which one representative is typically selected to perform the actual testing.

## 3.1 Input structure modeling

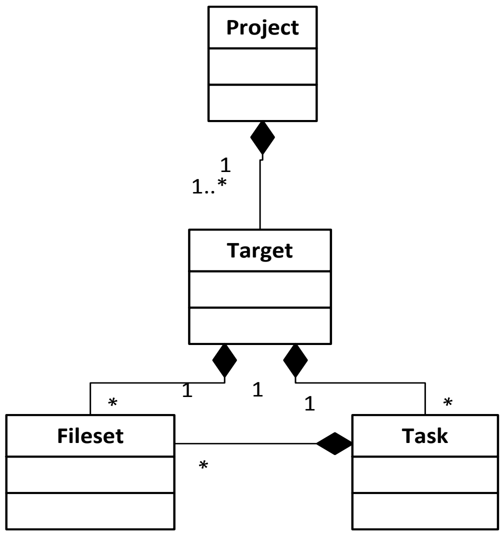
In this section, we focus on the graph structure. We consider two types of graph structure, depending on whether there is a loop in the graph structure.

### *3.1.1 Graph structure without loop*

As an example, consider the build XML file used by *Apache Ant*, which is shown below. A build XML file includes one project element and at least one target element. A target element includes one or more task elements. Each element has some attributes.

```
<project name="helloworld" basedir=".">
     <target name="compile">
            <mkdir dir="classes"/>
            <fileset file="main.java"/>
            <javac srcdir="src" destdir="classes"/>
    </target>
     <target name="jar">
            <mkdir dir="jar"/>
            <jar destfile="jar/HelloWorld.jar">
                      <fileset dir="classes"/>
            </jar>
    </target>
</project>
```

(a)



(b)

Figure 3.1 Example of graph structure without loop.
(a) A sample build XML file used by *Apache Ant* (b) The graph model of the sample build file

Figure 3.1(a) shows part of a build XML file used by Apache Ant. The file contains the "helloworld" project which has two targets. The first target "compile" has two task elements, "mkdir" and "javac", and one "fileset" element. The second target "jar" has two task elements, "mkdir" and "jar". The task element "jar" has a nested fileset element. Figure 3.1(b) shows the model of the example build file in Figure 3.1(a). We borrow some UML notations to depict the graph structure. These UML notations include class, attribute, multiplicity, constraint and composition relation notations.

The multiplicity notation shows that a project element has at least one target element. The composition relationship shows that the target and the task both use the fileset element. This suggests that we can model the fileset element once and then reuse it for both task and target elements. Note that the input structure is modeled as a graph even though the structure of the xml input file is a tree structure.

In the following, we give some guidelines on how to perform unit testing and integration testing based on a graph structure.

Figure 3.2(a) shows an example of a graph structure. Nodes A, B, and C are the parameters of the system and a1, a2, and a3 are the attributes of the node A and so on.

#### 3.1.1.1 Unit testing

Unit testing can be performed to test different combinations of attributes for individual nodes. In this case, attributes for the other required nodes will be given a default value. If an attribute has no default value, we will either exclude this attribute or will pick a value that we consider may be used most often.

Figure 3.2(b) shows an example of how to perform unit testing for the structure shown in Figure 3.2(a). Each node tests individually. For example node A has three binary attributes a1, a2, and a3 as its parameters. The 2-way test set for node A has four tests.

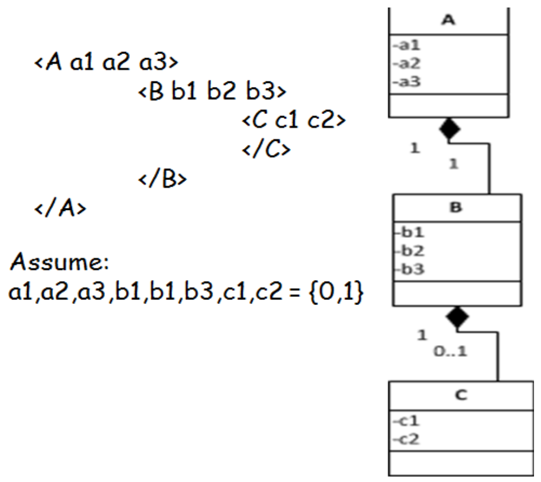
#### 3.1.1.2 Integration testing

Integration testing can be performed after unit testing of each node. The child node will be used as a composed parameter of the parent node.

Figure 3.2(c) and Figure 3.2(d) show two coverage options for the integration strategy. In both examples, node C is the first node to test. It has two binary attributes c1 and c2 as its parameters. The 2-way test set for node C has four tests. The second node to test is node B. Node B has three attributes with two values (binary attributes) and one nested element C. When we test node B, node C is also considered to be a new parameter of B, whose domain is the two-way test set derived earlier for unit testing of node C. That is, each test derived for unit testing of node C is considered to be one possible value node C could take as a parameter when we test node B.

We have two coverage options for parameter C depending on our domain knowledge. If there is no relationship between the attributes of the two nodes (B and C), then we do one-way coverage between the parameter C and the attributes of node B as shown in Figure 3.2(c)

The ACTS tool has a mixed relation feature that we can use to generate the mixed coverage between parameters. Otherwise, we perform t-way testing for all the parameters of node B, i.e., including its attributes and the new parameter added for node C as shown in Figure 3.2(d).



(a)

| B | | |
|---|---|---|
| b1 | b2 | b3 |
| 1 | 1 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 0 | 0 | 0 |

| A | | |
|---|---|---|
| a1 | a2 | a3 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 0 | 0 | 0 |
| 0 | 1 | 1 |

| C | |
|---|---|
| c1 | c2 |
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

(b)

| C | |
|---|---|
| c1 | c2 |
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

| B | | | |
|---|---|---|---|
| b1 | b2 | b3 | C |
| 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 2 |
| 0 | 1 | 1 | 3 |
| 0 | 0 | 0 | 4 |

| A | | | |
|---|---|---|---|
| a1 | a2 | a3 | B |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 2 |
| 0 | 0 | 0 | 3 |
| 0 | 1 | 1 | 4 |

(c)

| C | |
|---|---|
| c1 | c2 |
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

| B | | | |
|---|---|---|---|
| b1 | b2 | b3 | C |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 2 |
| 1 | 1 | 0 | 2 |
| 0 | 0 | 0 | 3 |
| 1 | 1 | 1 | 3 |
| 0 | 0 | 0 | 4 |
| 1 | 1 | 1 | 4 |

| A | | | |
|---|---|---|---|
| a1 | a2 | a3 | B |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 2 |
| 1 | 1 | 0 | 2 |
| 0 | 0 | 0 | 3 |
| 1 | 1 | 1 | 3 |
| 0 | 0 | 0 | 4 |
| 1 | 1 | 1 | 4 |
| 0 | 0 | 0 | 5 |
| ... | … | … | .. |

(d)

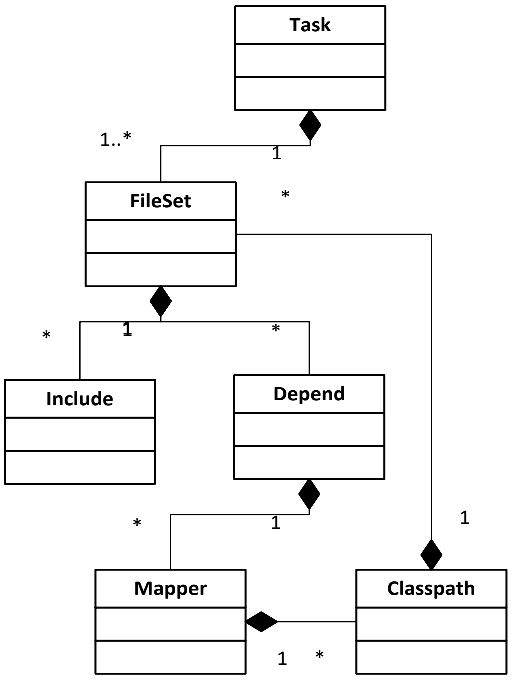
Figure 3.2 Structure based testing
(a) An example of a graph structure (b) Unit testing (c) Integration testing (mixed-coverage) (d) Integration testing (t-way coverage)

### 3.1.2 *Graph structure with loop*

Some graph structures may contain a loop. Figure 3.3 shows an example where there

```
<fileset dir="src" includes="*.java">
    <depend targetdir="/lib">
        <mapper classname="mapper.classname">
            <classpath>
                <fileset dir="lib">
                    <include name="*.class"/>
                </fileset>
            </classpath>
        <mapper>
    </depend>
</fileset>
```

(a)



(b)

Figure 3.3 Example of graph structure with loop
(a) A sample build XML file used by *Apache Ant* (b) The graph model of the sample build file

exists a loop involving nodes ‘classpath’, ‘fileset’, ‘depend’, and ‘mapper’. (The edges are directed in terms of the compositional relation.)

In the modeling process:

(1) Break the loop by removing the back edge. This is necessary to determine the order in which the nodes are going to be modeled. The back edge points from a node to another node that has already been visited during a DFS traverse.

(2) Model the graph without the loop (as we discussed in previous section).

(3) Place the back edge into the graph and add the destination node of the back edge into the model of the source node of the back edge.

In the above example, we break the loop by removing the back edge going from node 'classpath' to 'fileset'. After we modeled the graph without loop then we add the back edge and remodel the 'classpath' node. The 'classpath' node using the 'fileset' node as its nested element and we have already modeled the 'fileset' node which we can add to the model of 'classpath'.

Another important task is to generate the concrete test cases using the model. In order to do so we have to unfold the loop. We can unfold the loop once, more than once, or skip it. In this dissertation we only unfolded the loop once. The Figure 3.3 (a) shows the example of the concrete test case for the graph. The second 'fileset' with a nested 'include' is called inside the 'classpath'.

## 3.2 Input parameter modeling

After we model the input structure of the system and in order to perform testing, we need to model the input parameter of each node. The IPM method identifies the abstract model for the parameters, values, constraints, and relations from the specification. One common approach is to identify factors that could affect the behavior of the object being modeled. Each of these factors can be identified as an abstract parameter. Existing methods such as category partitioning and classification tree can be used to identify the abstract values of each parameter. The constraints are introduced to avoid invalid combinations. The relations are introduced to group parameters that are related to each other so that the different groups can be covered at different strengths.

For example the test factors for the jar task are shown in Table 3.1. This task will create a jarfile. In this table, the first column calls jar attributes. Note that not all the attributes of this element are always used as the test factors and also sometimes it is needed to add extra parameters.

There are three ways to specify the pattern of files that must be included or excluded, a list, a file, or a nested fileset. The *includes*/*excludes* attributes both are comma- or space-separated list of patterns that must be included or excluded. The *Includesfile*/*excludesfile* attribute specifies the name of a file which each line of this file is include/exclude pattern respectively. In addition we can use nested <fileset> as well. Two test factors are identified: include and exclude, with four abstract values [list, file, nested, off].

Moreover, the implicit *excludes/includes* attributes are used only if the value of *basedir* is 'on'. Therefore a constraint is identified.

Table 3.1 Jar task test factors

| Test factors | Test values |
|---|---|
| Destfile | [on, off] |
| Basedir | [on, off, empty] |
| Include | [list, file, nested, off] |
| Exclude | [list, file, nested, off] |
| Compress | [true,false] |
| Keepcompression | [true,false] |
| Update | [off,update,overwrite] |
| Destfileexist | [none, compressed, duplicate ] |
| Whenempty | [fail, skip, create] |
| Duplicate | [add, preserve, fail] |
| Filesetmanifest | [merge,skip,mergewithoutmain] |
| Nestedfileset | [NA, one, two or more] |

## 3.3 Derive concrete test cases

We use ACTS to generate t-way abstract test cases for each model. These test cases are abstract test cases because the parameters and values in the model are abstract. It is necessary to derive concrete test cases from these abstract test cases before testing is actually performed. Note that an abstract test case typically represents a set of concrete test cases, from which one representative is typically selected to perform the actual testing.

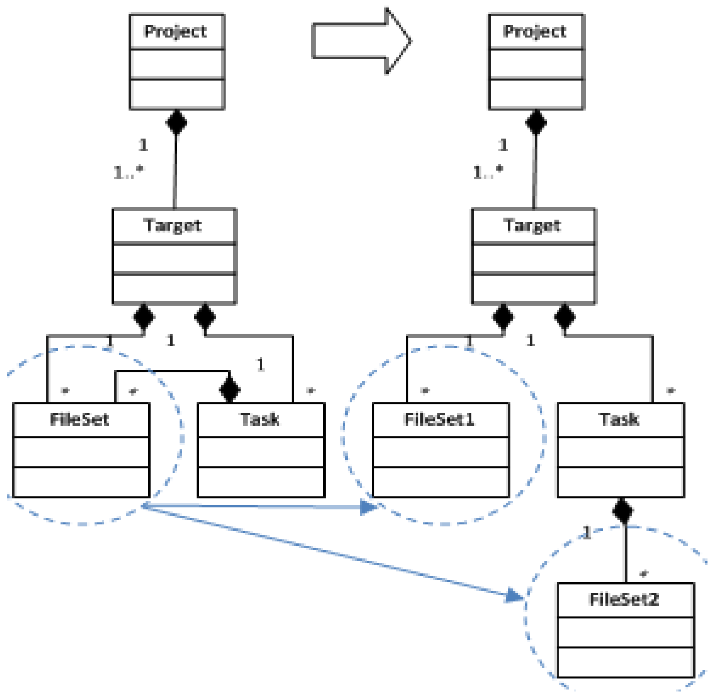
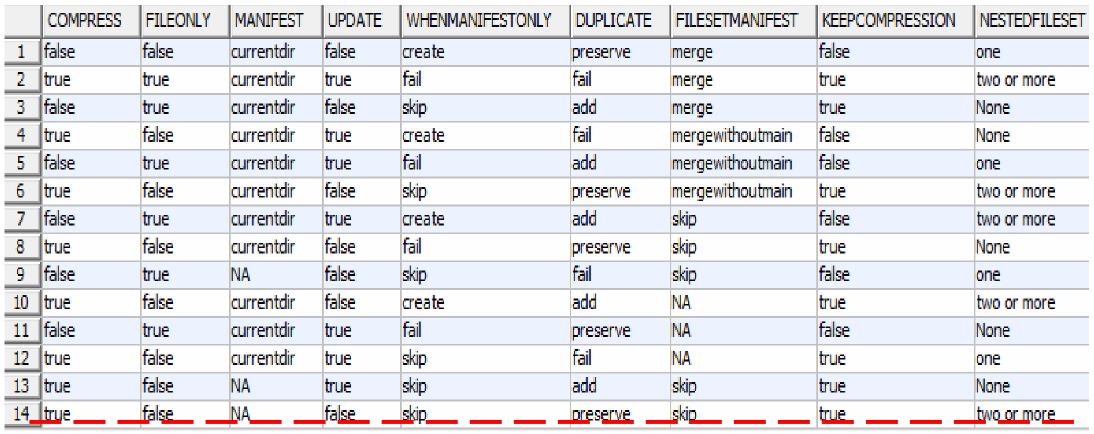


Figure 3.4 An example of node duplication

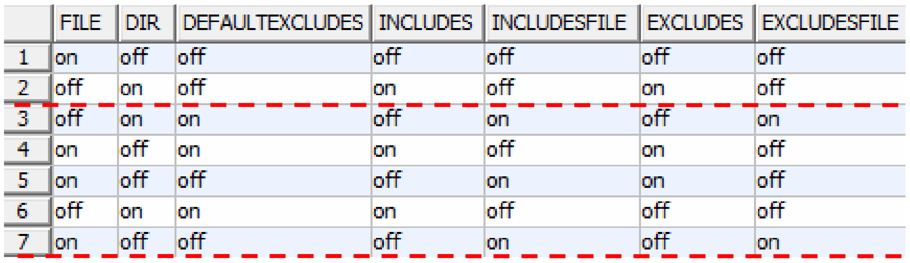
Since the graph structure represents the composition relation between different components, where one component may be composed of several other components, we often need to duplicate nodes that have more than one parent. Figure 3.4 shows an example of such duplication. The node ‘fileset’ has two parents meaning that ‘target’ and ‘task’ both will use the ‘fileset’. In order to generate the test, we need to duplicate the node ‘fileset’ to two nodes ‘fileset1’ and ‘fileset2’ such that each only has one parent. This does not mean that we model the ‘fileset’ node two times. Instead we reuse the same abstract model for the ‘fileset’ node to create two concrete values, which may be the same or different, to be used for the ‘task’ and ‘target’ nodes. The concept of auxiliary aggregate is to factor out the common parameters with identical concrete values in the model [27]. In contrast, we reuse the abstract model and the concrete values can be different.

We implemented a test case generator specific to each subject to automatically derive concrete values from the abstract test cases. Figure 3.5 and Figure 3.6 show the 2-way abstract test set for the ‘fileset’ and ‘jar’ tasks respectively. Each column represents a test factor (or abstract parameter) of the task and each row represents an abstract test case. The test case generator will select one representative for each test case and create a concrete test case.



| | COMPRESS | FILEONLY | MANIFEST | UPDATE | WHENMANIFESTONLY | DUPLICATE | FILESETMANIFEST | KEEPCOMPRESSION | NESTEDFILESET |
|---|---|---|---|---|---|---|---|---|---|
| 1 | false | false | currentdir | false | create | preserve | merge | false | one |
| 2 | true | true | currentdir | true | fail | fail | merge | true | two or more |
| 3 | false | true | currentdir | false | skip | add | merge | true | None |
| 4 | true | false | currentdir | true | create | fail | mergewithoutmain | false | None |
| 5 | false | true | currentdir | true | fail | add | mergewithoutmain | false | one |
| 6 | true | false | currentdir | false | skip | preserve | mergewithoutmain | true | two or more |
| 7 | false | true | currentdir | true | create | add | skip | false | two or more |
| 8 | true | false | currentdir | false | fail | preserve | skip | true | None |
| 9 | false | true | NA | false | skip | fail | skip | false | one |
| 10 | true | false | currentdir | false | create | add | NA | true | two or more |
| 11 | false | true | currentdir | true | fail | preserve | NA | false | None |
| 12 | true | false | currentdir | true | skip | fail | NA | true | one |
| 13 | true | false | NA | true | skip | add | skip | true | None |
| 14 | true | false | NA | false | skip | preserve | skip | true | two or more |

Figure 3.5 <jar> 2-way abstract test cases



| | FILE | DIR | DEFAULTEXCLUDES | INCLUDES | INCLUDESFILE | EXCLUDES | EXCLUDESFILE |
|---|---|---|---|---|---|---|---|
| 1 | on | off | off | off | off | off | off |
| 2 | off | on | off | on | off | on | off |
| 3 | off | on | on | off | on | off | on |
| 4 | on | off | on | on | off | on | off |
| 5 | on | off | off | off | on | on | off |
| 6 | off | on | on | on | off | off | off |
| 7 | on | off | off | off | on | off | on |

Figure 3.6 <fileset> 2-way abstract test cases

A sample concrete test for the test case number 14 in the Figure 3.5 is:

```
<jar destfile="test14.jar" compress="true" fileonly="false" update="false" duplicate="preseve"
keepcompression="true" >
            <fileset dir="/test" excludes=”*.jar" include=”*.class”/>
            <fileset dir="/test" defaultexcludes=”yes” include=”*.java” />
</jar>
```

In test case 14, the abstract value of the parameter nestedfileset is ‘two or more’. Therefore, in the concrete test case we include two nested <fileset> elements. The <fileset> elements are further selected from the <fileset> abstract test cases shown in Figure 3.6.

In addition, the test environment should contain a directory name ‘test’ and files with different extensions such as ‘.java’, ‘.class’ and ‘.jar’, in order for us to test the functionality of the above elements. Therefore, our test generator creates files with predefined extensions inside the ‘test’ directory.

# CHAPTER 4

# CASE STUDIES

We conducted several case studies in which the proposed methodology was applied to five real-life programs: *ACTS, Apache Ant*, *Space*, *Make* and *Grep*. The case studies are designed to answer three questions:

RQ1: Is the proposed methodology valid in a practical setting?

RQ2: How effective is combinatorial testing in terms of fault detection and coverage for real-life applications?

RQ3: How important is the modeling process for the effectiveness of combinatorial testing?

In our case studies, we compared combinatorial testing based on the proposed methodology to two random approaches. The first random approach, which is referred to as pure-random, generates random tests mainly based on the syntactic structure of the input space. The second random approach, which is referred to as modeled-random, generates random tests from the same model created by the proposed methodology.

We also considered the possibility of comparing our approach to an approach where we could create a model purely on the syntactic structure of the input space and then generate a t-way test set from this syntactical model. This approach requires less modeling effort, since no semantic information needs to be considered in the modeling process. The two approaches differ only in the modeling process. Thus this comparison would be good to show the difference caused by different modeling approaches. However, we found it was difficult to implement this approach. For example, we tried to apply this approach to the Apache Ant program and identified about 300 parameters. In addition, we had to introduce a large number of constraints to reflect the hierarchical structure. Thus we did not perform this comparison in our case studies.

For example if we modeled the small graph shown in Figure 3.2(a) with a non-structured method we would have to introduce more parameters, values and constraints as shown in Table 4.1. We had to add constraints in order to handle the hierarchy among the parameters.

Table 4.1 Non-structure based testing

| Test Factors | Test Values |
|---|---|
| A | [on, off] |
| Aa1 | [NA, 0,1] |
| Aa2 | [NA, 0,1] |
| Aa3 | [NA, 0,1] |
| B | [on, off] |
| Bb1 | [NA, 0,1] |
| Bb2 | [NA, 0,1] |
| Bb3 | [NA, 0,1] |
| C | [on, off] |
| Cc1 | [NA, 0,1] |
| Cc2 | [NA, 0,1] |

A='0ff' => B='off' && Aa1='NA' && Aa2='NA' && Aa3='NA'
B='0ff' => C='off' && Bb1='NA' && Bb2='NA' && Bb3='NA'
C='0ff' => Cc1='NA' && Cc2='NA'

## 4.1 Subject programs

Table 4.2 shows some statistics about the five subject programs. The programs are selected because of several desired attributes, including their complex input space, the existence of a clean version and multiple faulty versions, a relatively large number of lines of code, and the availability of their specifications.

Table 4.2 Statistics of subject programs

| Subject programs | LOC (Line of code) | # of classes/ Procedures | # of Faults in each faulty version | Type of faults (real or seeded) |
|---|---|---|---|---|
| ACTS | 24637 | 153 | 15 | Real |
| Ant 1.6 beta | 80500 | 627 | 6 | Seeded |
| Space | 9127 | 136 | 35 | Real |
| Make | 35545 | 268 | 19 | Seeded |
| Grep | 10068 | 146 | 5 | Real |

Subjects *Apache Ant*, *Space*, and *Make* are selected from the SIR repository [20] because SIR provides the clean version and multiple faulty versions for each of the subjects.

The types of faults are faults seeded for regression testing. The fault seeding process is explained in the SIR website. The fault seeders are programmers with at least two years of programming experience in Java/C/C++. There are at least two fault seeders for each subject. Each programmer must introduce faults independently to make the process as objective/credible as possible. They try to insert different faults such as faults associated with variables, control flow and memory allocation. The only requirement is that the changed program can still be compiled and executed. Faults that are considered too easy to be detected are removed.

Subject ACTS was conceived when a user of ACTS asked the question: Have you tested ACTS using ACTS? ACTS is a pretty mature, well documented, stable, and widely used tool for combinatorial test generation. The faulty version of ACTS was version 1.2 with bugs that were found in later versions. We used this version as the faulty version and version 1.4 (where the bugs were fixed) as the clean version. We applied our methodology to version 1.2 and compared the result of the faulty version with version 1.4.

#### *4.1.1 Overview of ACTS subject*

ACTS is a test generation tool for constructing *t*-way combinatorial test sets. Currently, it supports *t*-way test set generation with *t* up to 6. The tool is implemented in Java and provides both command line and graphical user interfaces. In the following, we briefly discuss the core features in ACTS.

- *T-Way Test Set Generation*: A system configuration is specified by a set of parameters and their values. A test set is a *t*-way test set if it satisfies the following property: Given any *t* parameters, every combination of values of these *t* parameters is covered in at least one test in the test set. Several test generation algorithms are implemented in ACTS. These algorithms include IPOG, IPOG-D, IPOG-F, IPOG-F2, and PaintBall. ACTS supports two test generation modes, namely, scratch and extend. The former allows a test set to be built from scratch, whereas the latter allows a test set to be built by extending an existing test set.

- *Mixed Strength* (*or Relation Support*): Relations are groups of parameters with different strengths. ACTS allows arbitrary parameter relations to be created, where different relations may overlap or subsume each other. In the latter case, relations that are subsumed by other relations will be ignored by the test generation engine.
- *Constraint Support*: Some combinations are not valid and must be excluded from the resulting test set. ACTS allows the user to define invalid combinations by specifying constraints. The specified constraints are taken into account during test generation so that the resulting test set will cover combinations that satisfy these constraints.
- *Coverage Verification*: This feature is used to verify whether a test set satisfies *t*-way coverage, i.e. whether it covers all the *t*-way combinations.

### *4.1.2 Overview of Ant subject*

*Apache Ant* is a software tool for automating software build processes. It is similar to Make but is implemented using the Java language, requires the Java platform, and is best suited to building Java projects. The most immediately noticeable difference between Ant and Make is that Ant uses XML to describe the build process and its dependencies, whereas Make uses Makefile format. By default the XML file is named build.xml.

Build.xml file contains one project and at least one target. Within each target are the actions that Ant must take to build that target; these are performed using built-in *tasks*. Each element of the build.xml file can have an *id* attribute which its value has to be unique.

### *4.1.3 Overview of Space subject*

*Space* performs as an interpreter for an Array Definition Language (ADL).The program reads an ADL format file that contains several ADL statements. The space language file was available instead of the specification. The file starts with a GROUP keyword follow by a group name, and ends with an END keyword.

The file contains of three main elements; grid and element definition, add and remove definitions and group excitation. The first element is an optional element but the other two are required.

The grid_element contains two parts. The first part related to GRID definition and the second part is the ELEMENT definitions and both of them are required. Also ELEMENT contains two optional parts: PORTS and POLARIZATION, and on required part: GEOMETRY.

In addition, the excitation element also contains of two required parts: PHASE and AMPLITUDE.

### *4.1.4 Overview of Make subject*

*Make* automatically determines which pieces of a large program need to be recompiled and issues commands to recompile them. To prepare to use make, we must write a file called the Makefile that describes the relationships among files in our program and provides command for updating each file. In a program, typically, the executable file is updated from object files, which are in turn made by compiling source files.

Once a suitable makefile exists, each time we change some source files, this simple shell command 'make' suffices to perform all necessary recompilations. The make program uses the makefile database and the last modification times of the files to decide which of the files need to be updated. For each of those files, it issues the recipes recorded in the database.

We can provide command line arguments to make to control which files should be recompiled, or how.

### *4.1.5 Overview of Grep subject*

Grep is a command-line utility for searching plain-text data sets for lines matching a regular expression. Based on the Grep specification, the program has three input parameters; pattern, input file and command line options. It reads an input file and search for a line or lines containing a match to a given pattern. There are zero or more options and also there can be zero or more input file names. A pattern is a regular expression that describes a set of strings. Grep understands three different versions of regular expression syntax: “basic,” (BRE), “extended” (ERE) and “perl”. In GNU Grep, there is no difference in available functionality between the basic and extended syntaxes. In other implementations, basic regular expressions are less powerful. We only limited out study to extended regular expressions. Perl regular expressions give additional functionality that may not be available on every system.

## 4.2 Overview of input models

Table 4.3 shows information about the input models built for each program. The second column shows number of models created for each program. The 3rd column shows the total number of parameters and their domain sizes in an exponential format, i.e., ($d_1^{p1}\ d_2^{p2}\ d_3^{p3} \cdots d_i^{pi}$), where $d_i^{pi}$ indicates that there are $p_i$ parameters with the domain size of $d_i$. Note that the total number of parameters is $\sum_{k=1}^{i} pk$. The 5th column represents the total number of constraints and the number of parameters involved in each constraint also in an exponential format, i.e., ($p_1^{c1}\ p_2^{c2}\ p_3^{c3} \cdots p_i^{ci}$), where $p_i^{ci}$ indicates that there are $c_i$ constraints with $p_i$ parameters. Note that the total number of constraints is $\sum_{k=1}^{i} ck$. The 7th column shows the total number of relations and the number of parameters involved in each relation also in an exponential format, i.e., ($p_1^{r1}\ p_2^{r2}\ p_3^{r3} \cdots p_i^{ri}$), where $p_i^{ri}$ indicates that there are $r_i$ relations with $p_i$ parameters. The total number of relations is $\sum_{k=1}^{i} rk$.

For example, *Space* has 7 models with 75 parameters. Five parameters have the domain size of 2, 43 parameters have the domain size of 3 and so on.

In the following sections we will discuss each model in more details.

Table 4.3 Input model summary

| Subject programs | Number of models | Total # of parameter values | Total # of parameters | Total # of constraints and the number of involved parameters | Total # of constraints | Total # of relations and the number of involved parameters | Total # of relations |
|---|---|---|---|---|---|---|---|
| ACTS | 19 | $2^{32}3^{10}4^{12}5^{3}6^{1}$ | 58 | $1^{2}2^{9}3^{3}5^{2}$ | 16 | $2^{6}3^{1}$ | 7 |
| Ant | 53 | $2^{122}3^{31}4^{14}6^{3}8^{2}$ | 172 | $1^{3}2^{29}3^{4}5^{2}6^{7}$ | 45 | $2^{8}3^{1}4^{1}$ | 10 |
| Space | 7 | $2^{5}3^{45}4^{20}5^{3}7^{2}$ | 75 | $2^{16}3^{14}4^{3}5^{3}6^{3}7^{3}9^{4}14^{2}18^{2}$ | 52 | $2^{14}3^{4}4^{2}6^{1}$ | 20 |
| Make | 7 | $2^{36}3^{9}4^{9}5^{1}10^{1}$ | 56 | $3^{3}$ | 3 | $3^{1}$ | 1 |
| Grep | 2 | $2^{32}3^{2}4^{7}9^{6}11^{1}$ | 48 | $2^{5}3^{6}4^{1}6^{13}$ | 25 | $2^{5}3^{2}5^{1}$ | 8 |

## 4.3 Test generation process

We used ACTS (version 2.7) [12] to generate t-way abstract test cases. We started from 2-way testing and then we extended the generated test cases to perform 3-way testing. As the number of test cases increases rapidly as the test strength increases, we did not go beyond 3-way testing.

We generated the modeled-random test cases by using the RANDBETWEEN(m,n) function of Microsoft Excel. The integer numbers (m and n) indicate the first and the last index of the parameter values. For example, consider a model of three parameters with domain size 4, RANDBETWEEN(1,4) generates a random integer between 1 and 4 for each parameter in a test case (Table 4.4). We used an IF function to check whether a test case satisfies all the constraints. In this study, we only used valid test cases, i.e., invalid tests were discarded.

For example, consider a model of three parameters with domain size 4, RANDBETWEEN(1,4) generates a random integer between 1 and 4 for each parameter in a test case (Table 4.4). We used an IF function to check whether a test case satisfies all the constraints. In this study, we only used valid test cases, i.e., invalid tests were discarded.

We used the sample xml generator *Oxygen* [26], to generate random XML files for the pure-random approach. *Oxygen* is an XML editor that creates XML documents based on a schema or a DTD file. It accepts a DTD file as input and converts it to a XML schema file (i.e., a

XSD file). It generates a user-defined number of random xml files from the schema. We set the number of repetitions and recursive levels to 2.

Table 4.4 Random abstract test case generation

| Parameters | P1 | P2 | P3 | valid/invalid |
|---|---|---|---|---|
| Formula | RAND BETWEEN(1,4) | RAND BETWEEN(1,4) | RAND BETWEEN(1,4) | =IF( AND(A2=B2,B2=C2), "invalid", "valid") |
| Test 1 | 1 | 4 | 1 | Valid |
| Test 2 | 3 | 3 | 3 | Invalid |

The *space* program takes as input a file in the ADL format. We wrote a program to convert the file from the ADL format to the XML format. Then, we generated random XML files using *Oxygen*, which are converted back to the ADL format. We followed the same approach for the *Make* program to convert a makefile to the XML format and vice versa.

The *Apache Ant* and *ACTS* programs take as input a XML file. The DTD for *Apache Ant* XML file is available. However, we did not model all the tasks[1] of the *Apache Ant*. Thus tasks that are not modeled in our approach were removed from the DTD file. We generated the schema file for the *ACTS* program using the *Oxygen* tool. The data in the XML file is used to infer a new XML Schema.

The *Grep* program has two inputs, pattern and input file. The program finds the pattern in the input file. The pattern is a regular expression. Some meta-characters have fixed positions in the pattern; they can just appear at the beginning or end of the pattern. But others can appear at the different positions of the pattern, beginning, middle or end. Based on the positions of the meta-characters, system may have different behavior. So instead of just having two values for these parameters, we define four values, *off*, *begin*, *middle* and *end*.

Table 4.5 shows the number of test cases generated for each subject program.

[1] We only modeled common tasks of *Apache Ant* such as archive, compile, documentation, exestuation, file tasks and logging tasks

Table 4.5 Number of test cases

| Testing method / Subject program | 2-way | 3-way |
|---|---|---|
| ACTS | 435 | 1105 |
| Ant | 836 | 2121 |
| Space | 120 | 315 |
| Make | 412 | 1230 |
| Grep | 358 | 910 |

## 4.4 Metrics

Two metrics are used to measure the effectiveness of each approach. The first metric is statement coverage. We use *clover* [7] to collect code coverage information for *Apache Ant* and *ACTS* which are written in java, and *gcov* [19] for *Space*, *Make* and *Grep* which are written in C.

The second metric is the number of faults detected by each approach. Each of the subject programs has multiple versions available: one clean version and several faulty versions. Each faulty version contains a single fault. We count the number of faulty versions that can be detected by each approach. Figure 4.1 shows a diagram of the fault detection procedure.

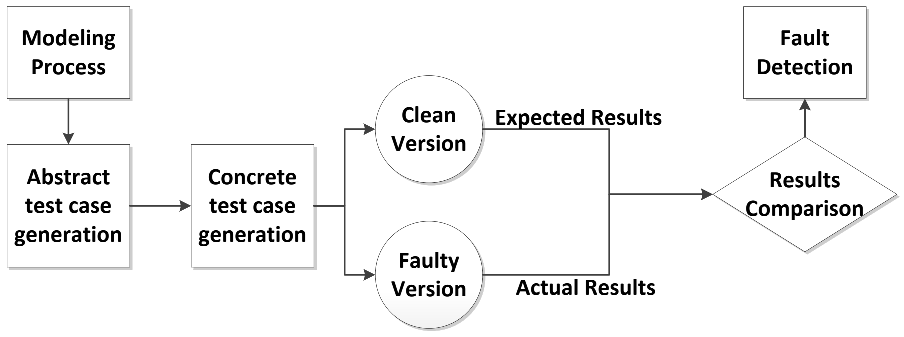


Figure 4.1 Fault detection procedure

## 4.5 Results and discussion

In this section we will explain the details of the input modeling, test case generation and results obtained for each subject program.

### *4.5.1 ACTS*

ACTS (Advanced Combinatorial Testing System) [12] is a combinatorial test generation tool developed jointly by the US National Institute Standards and Technology and the University of Texas at Arlington, and currently has more than 1200 individual and corporate users. This study was conceived when a user of ACTS asked the question: Have you tested ACTS using ACTS?

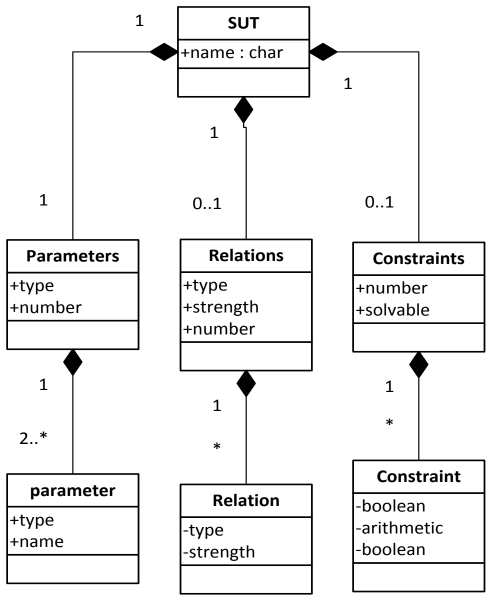


Figure 4.2 ACTS configuration file structure model

#### 4.5.1.1 Modeling for the System Under Test

A System Under Test (SUT) contains the configuration information of the system, including Parameters, Relations, and Constraints. In order to model the SUT, we have to model its components. The ACTS configuration file structure is shown in Figure 4.2. Since Parameters is one of the components of the SUT, in order to avoid confusion, we will use test factor instead of parameter throughout this dissertation. Also note that only for this subject we performed both robustness testing and functionality testing. For other subjects, as we mentioned before, we only performed functionality testing.

#### 4.5.1.1.1 Parameters

The parameters is defined to model the parameter component in the ACTS. The parameter itself has three parts; name, value, and type. Currently, four types of parameters are supported: Enum, Boolean, Range, and Integer. The Range type is basically a subset of Integer type. Entering a range is a feature in a GUI for facilitating entering values that are in range. It does not affect the system since it interprets to integer and then stores. However when we test the normal functionality of the GUI we consider the Range type as well. First, for each individual parameter, we identified two factors; number of values per parameter and type. The name factor is not important from the functionality perspective; therefore, we did not consider it in our model. The test factors for Parameter are shown in Table 4.6.

Table 4.6 Test factors of individual parameters

| Value per parameter | Type |
|---|---|
| invalid value | Integer |
| one valid value | range |
| Two or more valid values | Enum |
| | boolean |

Next, we discovered the relations between these factors. There are some constraints between type and value of a parameter. For example, the only valid values for a Boolean type parameter are [true, false]. For an Enum parameter, its value is either an invalid value such as a space character in robustness testing or a valid value in functionality testing. We want to ensure that for each parameter we cover all its type-value combinations at least once.

Table 4.7 Example parameter values

| Enum value | Integer or range value | Boolean value |
|---|---|---|
| special character | Small positive | [true , false] |
| numeric | Small negative | |
| alphabet | zero | |
| alphanumeric | Large positive | |
| Large string | Large negative | |

Some of these combinations are useful for functionality testing and others for robustness testing. The robustness testing for the command line interface and the graphical user interface (GUI) are different in some cases. For example, in the GUI when we select a Boolean type parameter, we cannot select any value, since its feature is disabled. This is not the case in the command line, where arbitrary values may be provided in a configuration file. Therefore, we applied the combination of Boolean type parameter with invalid value to perform robustness testing in the command line interface.

This model is an abstract model and we need concrete values to perform functionality testing. The Integer values were selected so that we have positive, zero, and negative values in our system. The values for the Boolean type are [true, false] by default. The values for the Enum type were selected so that we have a large and small number of values in our system. The Enum type in ACTS will accept any character but space. So we will use the space as an invalid value in robustness testing. The list of values for each parameter type is shown in Table 4.7.

Afterwards, multiple parameters are taken into account. Based on the ACTS specification, the system under test should have at least two parameters. Furthermore, the different types of parameters should be covered. The test factors for multiple parameters are shown in Table 4.8. We need to introduce some constraints to remove invalid combinations from our test results. For example if the number of parameters is two then only two parameter types can be set to 'on' and the others should be set to 'off'.

Table 4.8 Test factors of parameter group

| Number of parameters | Enum type | integer type | Range type | Boolean type |
|---|---|---|---|---|
| Invalid value (0 or 1) | on | on | on | on |
| Two | off | off | off | off |
| Three or more | | | | |

Finally, based on the obtained information, we generated executable test cases with concrete values. The following example is an abstract test case for the parameter group with three or more parameters which contains at least one parameter of each type:

```
Number of parameters: Three or more
Enum type= on
integer type= on
boolean type= on
range type= on
```

The concrete test case for the above abstract test contains seven parameters and covers all the parameter types at least once.

```
num1:[-1000, 10000]
num2:[-2, -1, 0, 1, 2]
bool1:[true,false]
bool2:[true, false]
Enum1:[v1, v2, v3, v4, v5, v6, v7, v8, v9]
Enum2:[1, 2]
Enum3:[#]
```

#### 4.5.1.1.2 Relations

The ACTS tool allows arbitrary relations between parameters to be created, where different relations may overlap or subsume each other or may subsume the default relation. First we identified test factors for individual relations. The ACTS tool has two types of relations; default and user-defined. ”Default” is the default relation of the system. This relation cannot be removed and it contains all of the system parameters and the current test strength. Also this relation will be automatically added to the system under test. The type and strength are the two test factors of the relation. Strength can be a number from 2 to 6 but we only performed our testing on 2, 3, and 6. 2 and 6 are boundary values. The test factors for individual relations are shown in Table 4.9.

Table 4.9 Test factors of individual relation

| Type | Strength |
|---|---|
| Default | 2 |
| User-defined (valid parameters) | 3 |
| User-defined (invalid parameters) | 6 |

Robustness testing for the command line interface and graphical user interface (GUI) is not the same in 'relations'. The user in the command line interface allows entering a relation to reference the parameters that do not exist in the system.

At this time, we identified the test factors of multiple relations. Based on the ACTS specification when the user adds the user-defined relations to the system, three different situations may occur. Because the default relation cannot be removed, the user-defined relations will always overlap with the default relation. They may also overlap with each other: "Overlap", or subsume each other: "Subsume", or subsume the default relation: "Subsume-default". The test factors for the user-defined relations are shown in Table 4.10.

Table 4.10 Test factors for user-defined relations

| Number of user-defined relations | Relation between user-defined and default relations |
|---|---|
| 0 | Overlap |
| 1 | Subsume |
| Two or more | Subsume the default |

Our goal was to cover all of the different relations in the system under test. When the number of user-defined relations is zero it means that the system contains only the "default" relation. When the number of user-defined relations is one this means that the system contains two relations; the default relation and the user-defined relation. In this condition, we introduced a user-defined relation that subsumes the default relation, "subsume-default". When the number of user-defined relations is two or more, the system contains three or more relations; the default relation and two or more user-defined relations. In this case, we introduced some user-defined relations that "subsume" or "overlap" each other to accomplish our goal.

Some examples of different relations in a system with the above mentioned values are shown in Table 4.11.

Table 4.11 Example relations

| Relation values | Example Relations [strength, (paramerename1, parametername2,..)] |
|---|---|
| Default | [4,(bool1, bool2, Enum1, Enum2, num1, num2)] |
| Subsume-default | [4,(bool1, bool2, Enum1, Enum2, num1, num2)] (default)<br>[5,(bool1, bool2, Enum1, Enum2, num1, num2)] |
| Overlap | [2,(bool1, bool2, Enum1)]<br>[2,(Enum1, Enum2, num1)] |
| Subsume | [3,(bool1, bool2, Enum1, Enum2, num1)]<br>[2,(bool1, bool2, Enum1, Enum2, num1)] |

The numbers in the bracket represent the strength while the symbols are the list of parameter names that interact with each other. The default strength in this example is 4. The second row shows a relation that subsumes the default relation in the first row. The third and fourth rows show the relations that overlap or subsume each other respectively.

4.5.1.1.3 Constraints

The constraints node is defined to model the constraint component in an SUT. Currently, three types of constraints are supported: Boolean, Relational, and Arithmetic. Each type will cover some symbols (operators) as shown in Table 4.12.

Table 4.12 Operators per constraint type

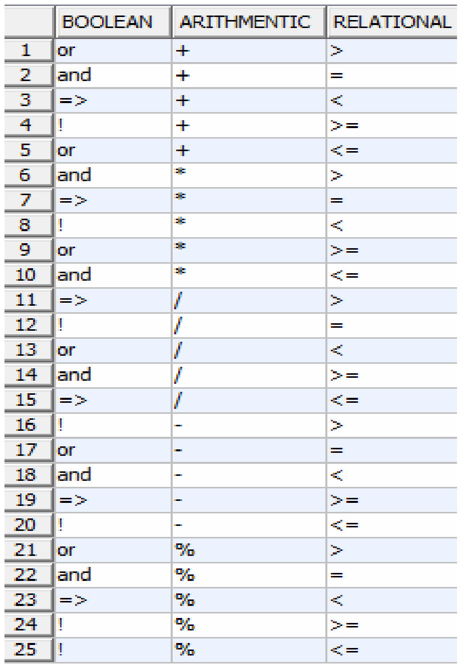
| Boolean | Arithmetic | Relational |
|---|---|---|
| Or | + | = |
| And | * | > |
| => | / | < |
| ! | - | ≥ |
|  | % | ≤ |

In order to have a meaningful constraint we need to generate a finite combination of symbols (operators) that are well-formed according to applicable rules. We used ACTS to generate all possible 2-way combinations between these three types of operators. ACTS generated 25 different combinations as shown in Table 4.13. For example three operators in the first row are or, +, and >. We manually generated a constraint that that covers all of them, e.g. p1+p2>1 or p3; p1 and p2 are two Integer type parameters and p3 is a Boolean type parameter.

We generated 25 different constraints to cover all the different 2-way combinations between the different types of constraints.

This model is an abstract model; we also need concrete values to perform testing. We used valid parameters to generate the constraints in normal functionality testing and one invalid parameter per constraint in robustness testing. An invalid parameter in this case is a parameter that is either not introduced to the system at all, or whose type does not match with its operator type, e.g. a Boolean type parameter and the arithmetic operator.

Table 4.13 2-way combinations of constraints types



| | BOOLEAN | ARITHMENTIC | RELATIONAL |
|---|---|---|---|
| 1 | or | + | > |
| 2 | and | + | = |
| 3 | => | + | < |
| 4 | ! | + | >= |
| 5 | or | + | <= |
| 6 | and | * | > |
| 7 | => | * | = |
| 8 | ! | * | < |
| 9 | or | * | >= |
| 10 | and | * | <= |
| 11 | => | / | > |
| 12 | ! | / | = |
| 13 | or | / | < |
| 14 | and | / | >= |
| 15 | => | / | <= |
| 16 | ! | - | > |
| 17 | or | - | = |
| 18 | and | - | < |
| 19 | => | - | >= |
| 20 | ! | - | <= |
| 21 | or | % | > |
| 22 | and | % | = |
| 23 | => | % | < |
| 24 | ! | % | >= |
| 25 | ! | % | <= |

Afterwards multiple constraints were taken into account. The test factors for multiple constraints are shown in Table 4.14.

We identified three factors for testing multiple constraints. The system under test can have zero, one, or multiple constraints. In addition, adding constraints to the system may introduce unsolvable constraints; therefore, the constraints are not always solvable.

Table 4.14 Test factors of multiple constraints

| Number of constraints | Relation between constraints | Satisfiability |
|---|---|---|
| 0 | Related | savable |
| 1 | Not_related | unsolvable |
| Multiple | | |

Furthermore, it is important to consider the relationship between different constraints. The constraints can be either related or not. The constraints are related if they share at least one parameter. The constraints are independent if they do not share any parameter.

The following example demonstrates the related constraints (Constraints 1 and 2 share the same parameter n2).

```
1.      (n2 >100) => !b2
2.      e1="1" => !(n2 >100)
```

The following example demonstrates independent constraints.

```
1.      (n2 >100) =>!b2
2.      e1="1" => !b1
```

These factors are independent and so we do not need to find the different combinations between them. However we need to consider them at least once during our testing process.

Finally based on the information obtained, we generated executable test cases with concrete values. The following example is an abstract test case for a system with three or more parameters, one parameter of each type, with multiple, related, and solvable constraints.

| | |
|---|---|
| Abstract test case: | Number of parameters: Three or more |
| | Parameter type: At least one parameter of each type |
| | Number of constraint: multiple |
| | Constraint relation: related |
| | Satisfiability: solvable |

The concrete test case for the above system is a system with six parameters and five solvable related constraints in which the constraints cover rows 2, 7, 15, 17, and 23 of Table 4.13:

```
num1:[-1000, 10000]
num2:[-2, -1, 0, 1, 2]
bool1:[true,false]
bool2:[true, false]
enum1:[v1, v2, v3, v4, v5, v6, v7, v8, v9]
enum2:[1, 2]

enum2="1" && num2+ num1=9999
(num1*num2= 1000) => bool1
num2/num1 <=500 => bool2
enum1="v1"|| num2-num1=9998
num1%num2<900 => num2<0
```

4.5.1.1.4 System under test

As we mentioned before System Under Test (SUT) contains the configuration information of the system parameters, relations and constraints. In the previous paragraphs we identified test values for each of these components; Parameters, Relations, and Constraints. We combined them to form the SUT model. The SUT factors and values are shown in Table 4.15.

We decided that there is no interaction between SUT factors; therefore, covering each value once would be sufficient. We produced the abstract model of SUT which is shown in Table 4.16. In total, 8 different system configurations have been identified for SUT, four of which were used in robustness testing.

Table 4.15 Test factors of SUT

| Test Factors | Test Values |
|---|---|
| Parameters | Invalid |
| | Two |
| | Three or more (at least one Integer, one Enum, one Boolean) |
| Relations | Invalid parameter (just in CMD interface) |
| | Default relation |
| | Two (default and subsume-default) |
| | Multiple relations (default plus at least two subsume) |
| | Multiple relations (default plus at least two overlap) |
| Constraints | None |
| | Unsolvable |
| | Invalid |
| | One |
| | Multiple not-related constraints |
| | Multiple related constraints |

Table 4.16 Abstract model of SUT

| Parameters | Relations | Constraints | SUT |
|---|---|---|---|
| Two | Two | Multiple not-related | 2Parameters_2Relations_multi-notrelatedConstraints |
| Multiple | Multiple | Multiple related | multiParameters_multiRelations_multirelatedConstraints |
| Multiple | Multiple | One | multiParameters_multiRelations_oneConstraint |
| Two | Default | None | 2Parameters_2Relations_noConstraint |
| Invalid | Default | One | Invalid Parameter |
| Two | Invalid | One | Invalid Relation |
| Two | Default | Invalid | Invalid Constraint |
| Two | Default | Unsolvable | Unsolved Constraint |

The following shows an example SUT with six parameters, multiple relations, and multiple related constraints:

**M_SUT with multiple parameters, multiple relations and multiple related Constraints (multiParameters_multiRelations_multi-relatedConst)**

```
  Default degree of interaction coverage: 4
  Number of parameters: 6
Parameters:
  num1:[-1000, -100, 1000, 10000]
  num2:[-2, -1, 0, 1, 2]
  bool1:[true, false]
   bool2:[true, false]
   Enum1:[v1, v2, v3, v4, v5, v6, v7, v8, v9]
   Enum2:[1, 2]
Relations :
  [4,(bool1, bool2, Enum1, Enum2, num1, num2)]
  [5,(bool1, bool2, Enum1, Enum2, num1, num2)]
   [2,(bool1, bool2, Enum1)]
  [2,(Enum1, Enum2, num1)]
   [3,(bool1, bool2, Enum1, Enum2, num1)]
Constraints :
  enum2="1" && num2+ num1=9999
  (num1*num2= 1000) => bool1
  num2/num1 <=500 => bool2
  enum1="v1"|| num2-num1=9998
   num1%num2<900 => num2<0
```

The following is an example SUT with “Invalid constraint”:

**M_SUT with invalid constraint (num3 doesn’t exist)**

```
  Default degree of interaction coverage: 4
  Number of parameters: 6
  Number of configurations: 0
Parameters:
  num1:[-1000, -100, 1000, 10000]
  num2:[-2, -1, 0, 1, 2]
  bool1:[true, false]
   bool2:[true, false]
   Enum1:[v1, v2, v3, v4, v5, v6, v7, v8, v9]
   Enum2:[1, 2]
Relations :
  [4,(bool1, bool2, Enum1, Enum2, num1, num2)]
  [5,(bool1, bool2, Enum1, Enum2, num1, num2)]
Constraints :
  (num1*num2>num2+100) => bool2!=bool1
  num2/num1 >=10 => !bool2
  num1%num2<=3 => num1<4
  bool1 =>Enum1="v1"
  Enum2="1" && Enum1="v2" => num2=2 || num3=0
```

The factors discussed in the above paragraphs are common between two different interfaces of ACTS. The following paragraphs, however, will identify the factors and values that are different between the command line interface and the GUI interface.

#### 4.5.1.2 Modeling for the Command Line Interface

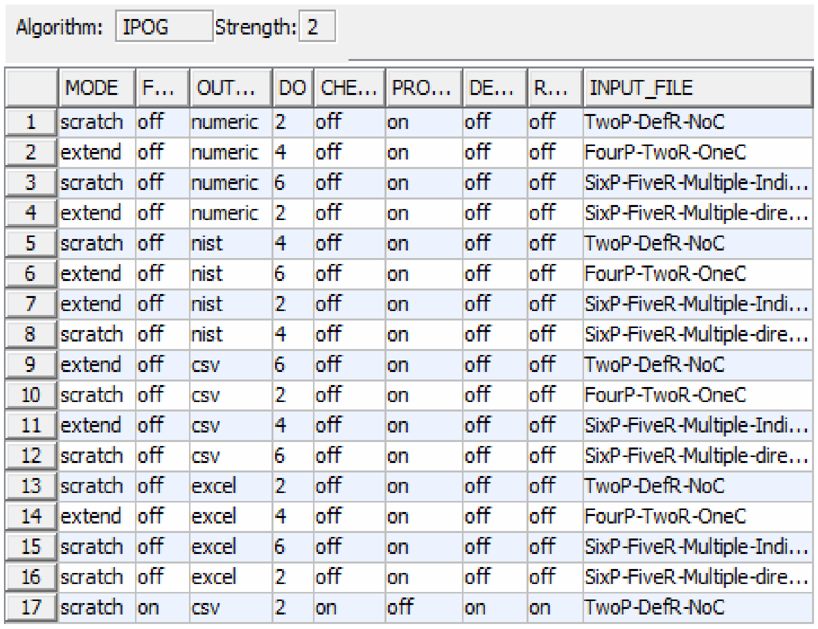
The various options are available in the command line interface as shown in Table 4.17. There are several test generation algorithms implemented in ACTS. The user has to select one of these algorithms in order to generate the tests. “Algorithm” would be chosen as one of our factors with the domain value of [IPOG, IPOG-D, IPOG-F, IPOG-F2, PaintBall]. The IPOG algorithm is the most commonly used algorithm; therefore, in this case study we performed our test on the IPOG and fixed the value of Algorithm to “IPOG”. Covering other algorithms will be our future work. Also, ACTS supports two test generation modes, scratch and extend. Obviously “mode” is another factor with the domain of [scratch, extend].

Table 4.17 Command line model

| Test Factors | Test Values | Description |
|---|---|---|
| Mode | Scratch | generate tests from scratch (default) |
| | Extend | extend from an existing test set |
| Algorithm | Ipog | use algorithm IPO (default) |
| fastMode | On | enable fast mode |
| | Off | disable fast mode (default) |
| Doi | specify the degree of interactions to be covered | |
| Output | Numeric | output test set in numeric format |
| | Nist | output test set in NIST format (default) |
| | Csv | output test set in Comma-separated values format |
| | Excel | output test set in EXCEL format |
| Check | On | verify coverage after test generation |
| | Off | do not verify coverage (default) |
| Progress | On | display progress information (default) |
| | Off | do not display progress information |
| Debug | On | display debug info |
| | Off | do not display debug info (default) |
| Randstar | On | randomize don't care values |
| | Off | do not randomize don't care values |

Some of these options e.g. fastmode, check, debug, randstar, and progress are totally independent from each other. Because there is no interaction between them they only need to appear in the test once.

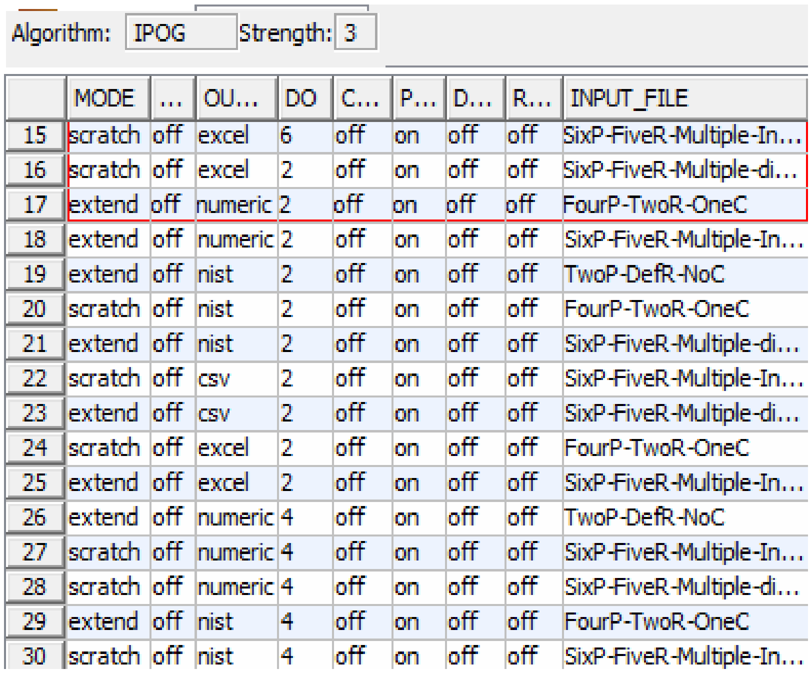
Figure 4.3(a) shows the test cases generated by ACTS for the command line interface with test strength t=2. We extended it to t=3 to see whether we could detect more faults. Figure 4.3(b) shows some of the test cases generated by ACTS for t=3.



Algorithm: IPOG Strength: 2

| | MODE | F... | OUT... | DO | CHE... | PRO... | DE... | R... | INPUT_FILE |
|---|---|---|---|---|---|---|---|---|---|
| 1 | scratch | off | numeric | 2 | off | on | off | off | TwoP-DefR-NoC |
| 2 | extend | off | numeric | 4 | off | on | off | off | FourP-TwoR-OneC |
| 3 | scratch | off | numeric | 6 | off | on | off | off | SixP-FiveR-Multiple-Indi... |
| 4 | extend | off | numeric | 2 | off | on | off | off | SixP-FiveR-Multiple-dire... |
| 5 | scratch | off | nist | 4 | off | on | off | off | TwoP-DefR-NoC |
| 6 | extend | off | nist | 6 | off | on | off | off | FourP-TwoR-OneC |
| 7 | extend | off | nist | 2 | off | on | off | off | SixP-FiveR-Multiple-Indi... |
| 8 | scratch | off | nist | 4 | off | on | off | off | SixP-FiveR-Multiple-dire... |
| 9 | extend | off | csv | 6 | off | on | off | off | TwoP-DefR-NoC |
| 10 | scratch | off | csv | 2 | off | on | off | off | FourP-TwoR-OneC |
| 11 | extend | off | csv | 4 | off | on | off | off | SixP-FiveR-Multiple-Indi... |
| 12 | scratch | off | csv | 6 | off | on | off | off | SixP-FiveR-Multiple-dire... |
| 13 | scratch | off | excel | 2 | off | on | off | off | TwoP-DefR-NoC |
| 14 | extend | off | excel | 4 | off | on | off | off | FourP-TwoR-OneC |
| 15 | scratch | off | excel | 6 | off | on | off | off | SixP-FiveR-Multiple-Indi... |
| 16 | scratch | off | excel | 2 | off | on | off | off | SixP-FiveR-Multiple-dire... |
| 17 | scratch | on | csv | 2 | on | off | on | on | TwoP-DefR-NoC |

(a)



Algorithm: IPOG Strength: 3

| | MODE | ... | OU... | DO | C... | P... | D... | R... | INPUT_FILE |
|---|---|---|---|---|---|---|---|---|---|
| 15 | scratch | off | excel | 6 | off | on | off | off | SixP-FiveR-Multiple-In... |
| 16 | scratch | off | excel | 2 | off | on | off | off | SixP-FiveR-Multiple-di... |
| 17 | extend | off | numeric | 2 | off | on | off | off | FourP-TwoR-OneC |
| 18 | extend | off | numeric | 2 | off | on | off | off | SixP-FiveR-Multiple-In... |
| 19 | extend | off | nist | 2 | off | on | off | off | TwoP-DefR-NoC |
| 20 | scratch | off | nist | 2 | off | on | off | off | FourP-TwoR-OneC |
| 21 | extend | off | nist | 2 | off | on | off | off | SixP-FiveR-Multiple-di... |
| 22 | scratch | off | csv | 2 | off | on | off | off | SixP-FiveR-Multiple-In... |
| 23 | extend | off | csv | 2 | off | on | off | off | SixP-FiveR-Multiple-di... |
| 24 | scratch | off | excel | 2 | off | on | off | off | FourP-TwoR-OneC |
| 25 | extend | off | excel | 2 | off | on | off | off | SixP-FiveR-Multiple-In... |
| 26 | extend | off | numeric | 4 | off | on | off | off | TwoP-DefR-NoC |
| 27 | scratch | off | numeric | 4 | off | on | off | off | SixP-FiveR-Multiple-In... |
| 28 | scratch | off | numeric | 4 | off | on | off | off | SixP-FiveR-Multiple-di... |
| 29 | extend | off | nist | 4 | off | on | off | off | FourP-TwoR-OneC |
| 30 | scratch | off | nist | 4 | off | on | off | off | SixP-FiveR-Multiple-In... |

(b)

Figure 4.3 Command line test cases

(a) Test cases with t=2 (b) Part of test cases with extend t=2 to t=3

### 4.5.1.3 Modeling for the Graphical User Interface

ACTS is a complex system with several features and functionalities. The divide-and-conquer strategy is used to model the GUI. We divided the system based on the system use-cases. The use-cases are often used to capture the system functionalities. We derived the ACTS's use-cases from the user document and captured several features for the GUI such as Create New System, Build Test Set, Modify System (add/remove/edit parameters and parameters values, add/remove relations, add/remove constraints), Open/Save/Close System, Import/Export Test Set, Statistics, and Verify Coverage. For each of these we designed a separate IPM to yield several small IPMs rather than one large one.

#### 4.5.1.3.1 Modify system

Modification is the process of changing a system configuration. Designing the IPM for this feature was very challenging because this feature has several functionalities. We divided the modification feature to the following smaller IPMs: Add Parameter, Remove Parameter, Modify parameter, Add Constraint, Remove constraint, Add Relation, and Remove Relation.

#### 4.5.1.3.2 Add a parameter

First, in order to add a parameter, the user has to enter a parameter name to activate the add button. We call this name in the model with [valid, invalid] test values. The user may enter space or a special character or number but these are invalid and the system will show the related error messages.

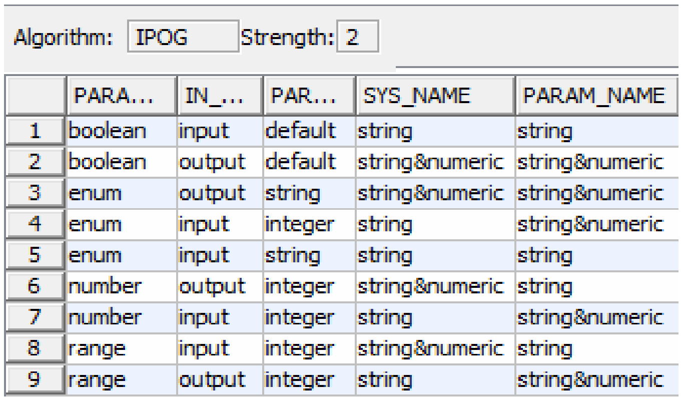
The only acceptable name is string without any space. Next selecting a type and entering values for the parameter 'value'; also these parameters can be input parameters or output (in_out). In addition if the type of the parameter is "Boolean" then the user cannot enter any value because the system has two default values for the Boolean types [true,false], also if type is "Range" the user cannot enter any value but selecting the range. [-9000000 to 9000000] could be an invalid range. Basically these are some invalid combinations between the type and the value which we have to exclude from the final test cases. As mentioned before, ACTS has a constraint support feature so we can add the following constraint. The model of “add parameter” is shown in Table 4.18. The test cases with valid values generated by ACTS with strength t=2 are shown in Figure 4.4. Also test cases containing invalid values are shown in Figure 4.5.

| |
|---|
| type="Boolean” => value="Default" |

This means that if the type of the parameter is “Boolean”, the system will fix the value to “default”.

Table 4.18 GUI, add parameter model

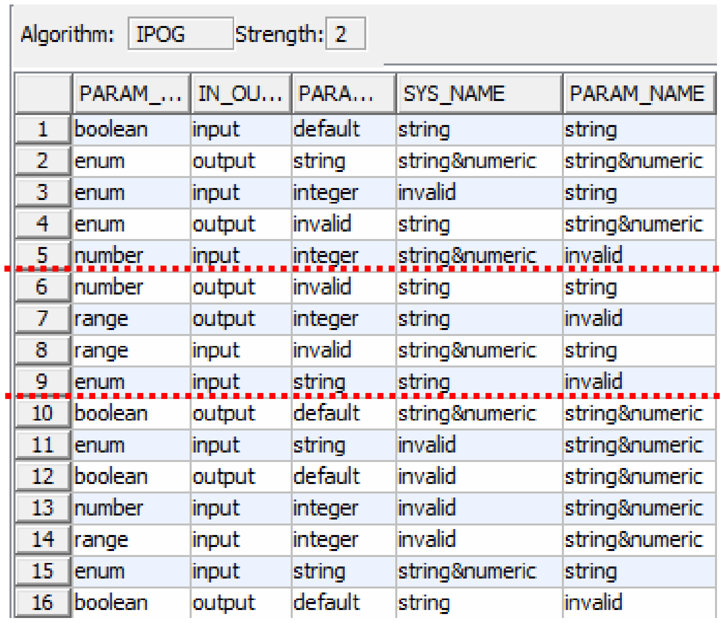
| Test Factors | Test Values |
|---|---|
| sys_name | invalid (space, special_char, number, duplicate name) |
| | String only |
| | String plus numeric |
| Name | invalid (space, special_char, number, duplicate name) |
| | String only |
| | String plus numeric |
| Type | Boolean |
| | Enum |
| | Number |
| | Range |
| in_out | Input |
| | Output |
| Value | Integer |
| | String |
| | Default |
| | Invalid (Space, duplicate value, invalid range of numbers or characters) |



Algorithm: IPOG Strength: 2

| | PARA... | IN_... | PAR... | SYS_NAME | PARAM_NAME |
|---|---|---|---|---|---|
| 1 | boolean | input | default | string | string |
| 2 | boolean | output | default | string&numeric | string&numeric |
| 3 | enum | output | string | string&numeric | string&numeric |
| 4 | enum | input | integer | string | string&numeric |
| 5 | enum | input | string | string | string |
| 6 | number | output | integer | string&numeric | string |
| 7 | number | input | integer | string | string&numeric |
| 8 | range | input | integer | string&numeric | string |
| 9 | range | output | integer | string | string&numeric |

Figure 4.4 Test cases of add parameter with valid values



Algorithm: IPOG Strength: 2

| | PARAM_... | IN_OU... | PARA... | SYS_NAME | PARAM_NAME |
|---|---|---|---|---|---|
| 1 | boolean | input | default | string | string |
| 2 | enum | output | string | string&numeric | string&numeric |
| 3 | enum | input | integer | invalid | string |
| 4 | enum | output | invalid | string | string&numeric |
| 5 | number | input | integer | string&numeric | invalid |
| 6 | number | output | invalid | string | string |
| 7 | range | output | integer | string | invalid |
| 8 | range | input | invalid | string&numeric | string |
| 9 | enum | input | string | string | invalid |
| 10 | boolean | output | default | string&numeric | string&numeric |
| 11 | enum | input | string | invalid | string&numeric |
| 12 | boolean | output | default | invalid | string&numeric |
| 13 | number | input | integer | invalid | string&numeric |
| 14 | range | input | integer | invalid | string&numeric |
| 15 | enum | input | string | string&numeric | string |
| 16 | boolean | output | default | string | invalid |

Figure 4.5 Test cases of add parameter contains invalid values

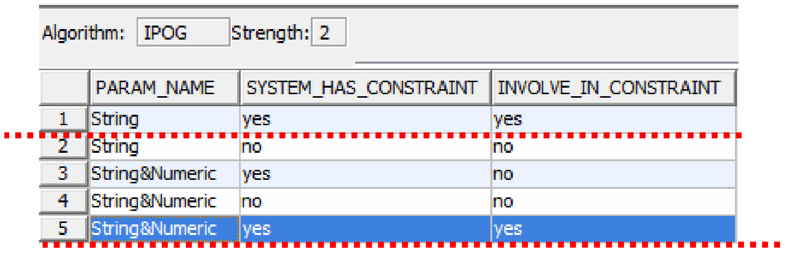
4.5.1.3.3 Change parameter name

The user can change the name of a parameter. The new name should be a valid name (no space). This parameter should not be involved in any constraint; otherwise the name has to be changed automatically everywhere in the system in which this parameter is used. The model of “change parameter name” is shown in Table 4.19. The test cases with valid values generated by ACTS with strength t=2 are shown in Figure 4.6.

The Involved_in_constraint is a factor to guarantee we will test parameters that are involved in a constraint.

Table 4.19 GUI, change parameter name IPM

| Test Factors | Test Values |
|---|---|
| Name | String only |
|  | String plus numeric |
|  | Invalid (space, special_char, number, duplicate name) |
| Involved_in_constraint | [Yes,no] |
| System_has_constraint | [Yes,no] |


Algorithm: IPOG Strength: 2

|  | PARAM_NAME | SYSTEM_HAS_CONSTRAINT | INVOLVE_IN_CONSTRAINT |
|---|---|---|---|
| 1 | String | yes | yes |
| 2 | String | no | no |
| 3 | String&Numeric | yes | no |
| 4 | String&Numeric | no | no |
| 5 | String&Numeric | yes | yes |

Figure 4.6 Test cases of change parameter name

4.5.1.3.4 Build system

The model of build system is shown in Table 4.20 and Table 4.21. All of the parameters are discussed earlier. Valid IPM is used to test the normal functionality of the system and invalid IPM is used for robustness testing.

Table 4.20 GUI, build valid model

| Test Factors | Test Values |
|---|---|
| Mode | [Scratch, Extend |
| Algorithm | [IPOG] |
| Strength | [2,4,6] |
| Randomize | [On,off] |
| Progress | [On,off] |
| SUT | [2P_2R_multi-nC, multiP_multiR_multi-rC, multiP_multiR_multi-rC, multiP_multiR_oneC, 2P_2R_noC] |

Table 4.21 GUI, build invalid model

| Test Factors | Test Values |
|---|---|
| mode | [Scratch, Extend] |
| algorithm | [IPOG] |
| Strength | [2,4,6] |
| randomize | [On,off] |
| progress | [On,off] |
| SUT | [InvalidP, InvalidR, InvalidC, UnsolvedC] |

After we created our models, we used them as an input to the ACTS tool, which will give us all the combinations between factors for each model. The number of models was 19.

We integrated the smaller IPMs together using an interaction-based test sequence generation to completely test the system. The reason we decided to use this method was that some of the bugs would not be triggered by just testing each use-case individually. It is important to test a sequence of events in order to test the whole system completely. Wenhua Wang et al. [6] present a test sequence generation approach for covering all interactions between any two pages of a web application.

We can generalize this algorithm to be able to use it in combinatorial testing of systems with a GUI as well. First we generated a navigation graph of our use cases. There exists an edge from one node m to another node n if node n can be visited immediately after node m through a direct link. Each node is a use-case. A simplified form of ACTS's use-cases navigation graph is shown in Figure 4.7. Using the navigation graph was very helpful because not all the combinations between the use-cases are feasible. The graph helped to visualize the feasible and infeasible sequences.

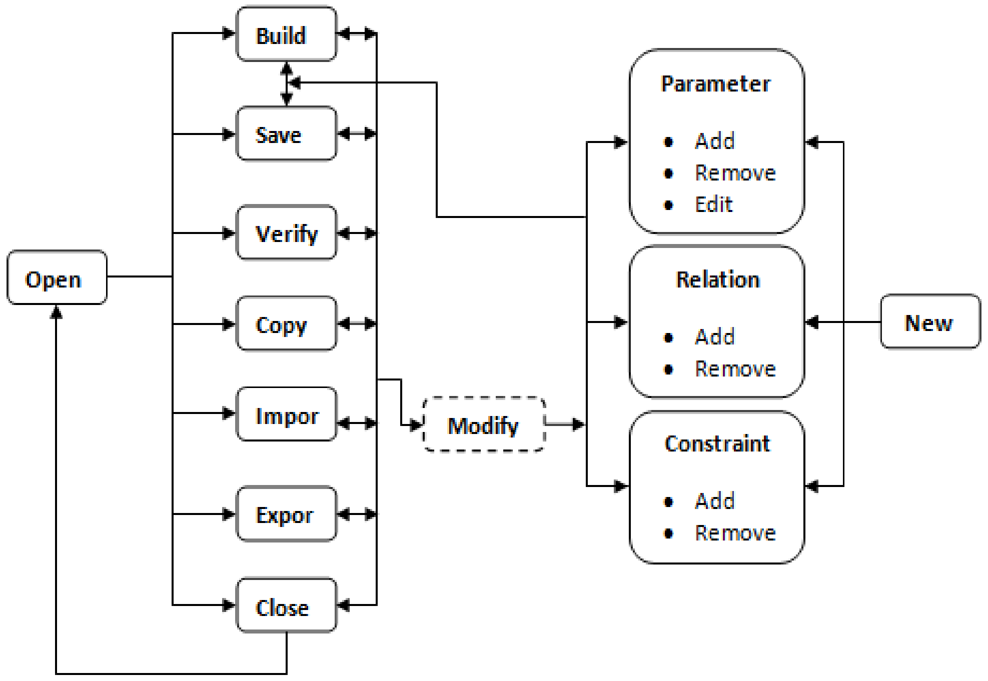


Figure 4.7 ACTS's navigation graph

Next we generated a test sequence to satisfy pairwise interaction coverage. The term "pairwise interaction" refers to interaction between two nodes. Let G = (V, E, n0) be a navigation graph. Formally, a pairwise interaction in G is an ordered pair (m, n), where m and n are two nodes, and there exists a path from m to n in G. Pairwise interaction coverage requires that a set of paths be selected from a navigation graph as test sequences so that every ordered pair is covered in at least one of those test sequences. We generated all of the ordered pairs for use-cases from the navigation graph.

In following sections we provide some examples of the sequences that lead us to find the faults in ACTS. In this case study we limited the length of sequences to be six. In future work we will use tools such as *GUI Ripper* to remove human error in this part of the work [16].

The design model for ACTS has 19 IPMs which are shown in Table 4.22, yielding 1105 generated test cases. Number of uncommented lines of code in ACTS are 24637.

Table 4.22 Summary of ACTS input model

| ACTS | # of parameter values | # of parameter | # of constraint parameters | # of constraint | # of relation parameters | # of relation |
|---|---|---|---|---|---|---|
| CMD | $2^4 4^3$ | 7 | $1^1 3^1 5^1$ | 3 | $2^2$ | 2 |
| Build | $2^3 4^2 6^1$ | 6 | $1^1 5^1$ | 2 | $2^2$ | 2 |
| New system | $2^1 3^1 4^2$ | 4 | $2^2$ | 2 | $2^1 3^1$ | 2 |
| Add Parameter | $2^2 3^1 5^1$ | 4 | $2^3$ | 5 | $2^1$ | 1 |
| Remove parameter | $2^2$ | 2 | - | 0 | - | 0 |
| Change Name | $2^2$ | 2 | $2^1$ | 1 | - | 0 |
| Add value | $3^2 5^1$ | 3 | $2^1$ | 1 | - | 0 |
| Remove value | $2^2 3^1$ | 3 | $3^2$ | 2 | - | 0 |
| Add relation | $2^1 4^1$ | 2 | - | 0 | - | 0 |
| Remove relation | $2^1 4^1$ | 2 | - | 0 | - | 0 |
| Add constraint | $2^1 3^2$ | 3 | - | 0 | - | 0 |
| Remove Constraint | $2^1 5^1$ | 2 | - | 0 | - | 0 |
| Open | $2^1 3^1 4^1$ | 3 | - | 0 | - | 0 |
| Close | $2^2$ | 2 | - | 0 | - | 0 |
| Verify | $2^2$ | 2 | - | 0 | - | 0 |
| Import | $2^2 4^1$ | 3 | - | 0 | - | 0 |
| Export | $2^1 4^1$ | 2 | - | 0 | - | 0 |
| Save | $2^3$ | 3 | - | 0 | - | 0 |
| Statistics | $2^1 3^2$ | 3 | - | 0 | - | 0 |
| TOTAL | $2^{32} 3^{10} 4^{12} 5^3 6^1$ | 58 | $1^2 2^9 3^3 5^2$ | 16 | $2^6 3^1$ | 7 |

4.5.1.4 Results and discussion

The design model for ACTS has 19 valid IPMs which are shown in Table 4.22, yielding 1105 generated test cases. Code coverage data are shown in Figure 4.8 and Figure 4.9. We used *clover* to collect code coverage [7]. We ran *clover* with eclipse and executed our tests on ACTS version 1.2. ACTS statistics are shown in Table 4.23. e.g. number of uncommented lines of code in ACTS are 24637.

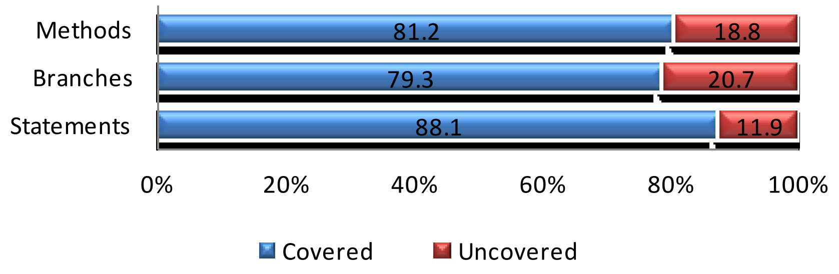


Figure 4.8 ACTS effectiveness metrics

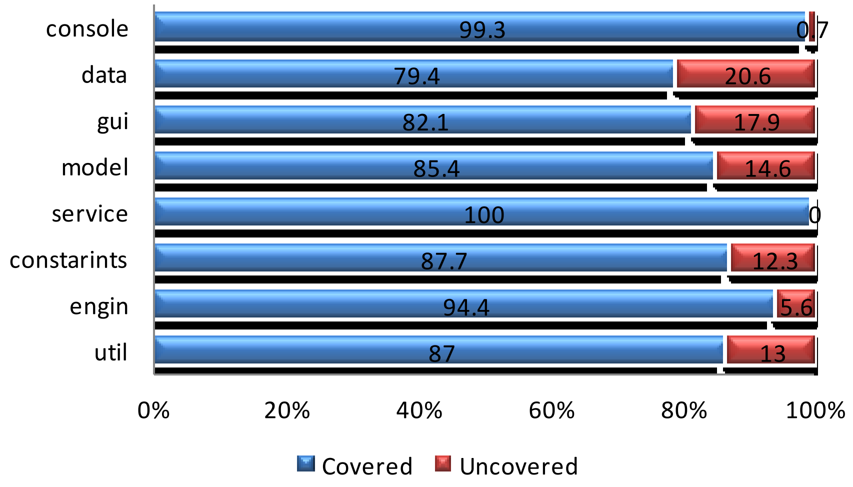


Figure 4.9 Code coverage for ACTS packages

Clover gave us the code coverage for all of the test cases. While we executed our tests, clover highlighted the parts of the source code that were executed. This made it easy to identify the code that was never called during our testing process. It is shown in Figure 4.8 that our tests covered more than 88% of system statements. Figure 4.9 shows different packages of ACTS. We covered 99% the Console package.

Table 4.23 ACTS statistics

| | |
|---|---|
| LOC (line of code) | 38,165 |
| NC LOC | 24,637 |
| Number of Statements | 13,642 |
| Number of Branches | 4,696 |
| Number of Methods | 1,693 |
| Number of Classes | 153 |
| Number of Files | 110 |
| Number of Packages | 12 |

Other packages are more related to the GUI. Packages, e.g. Engine, Model, Util, GUI, and Data are common between different algorithms. We only performed testing on the "IPOG" algorithm. There are five more algorithms implemented in ACTS. Therefore, we have not exercised some statements in our case studies.

We classified detected faults in ACTS into four groups as shown in Table 4.24. The First group is the faults related to functionality testing of graphical user interface. The second group is the faults related to robustness testing of graphical user interface. The third group is the faults related to functionality testing of command line interface. The fourth group is the faults related to robustness testing of command line interface.

Table 4.24 Faults classification

| Fault Groups | Number of Detected Faults |
|---|---|
| functionality testing of GUI | 10 |
| robustness testing of GUI | 5 |
| functionality testing of cmd | 1 |
| robustness testing of cmd | 1 |

The total number of detected faults is 15, 10 of which detected by functionality testing and 5 of them detected by robustness testing. In our case study some of the faults detected in the GUI occurred in the command line interface as well. One possible reason that we only detect 15 faults out of almost 1000 tests is because the ACTS is pretty mature software, well documented, stable, and widely used, Also some of the detected bugs are single mode faults.

The results of pure-random testing and modeled-random testing on ACTS subject are shown in Figure 4.10 and Table 4.25. We only perform the testing on the command line interface of ACTS subject. To perform the pure-random testing we used Oxygen. Oxygen uses the DTD file to generate random xml file. Since the schema file or DTD file of ACTS input file was not available, we used Oxygen to convert the xml files to a DTD file. The problem was that when ACTS generates a parameter and its value(s), it uses them to generate relations and constraints. But inside the DTD file, there was no unique id for the parameters and thus it would generate the new parameter each time. Therefore all the random xml files generated with this DTD file were failed to run. We modified the DTD file and added the unique id and reference for the parameter name and value to the DTD file. In addition some of the attributes such as strength only get values between 2 to 6. Therefore we restricted the random number to be in range 2 to 6.

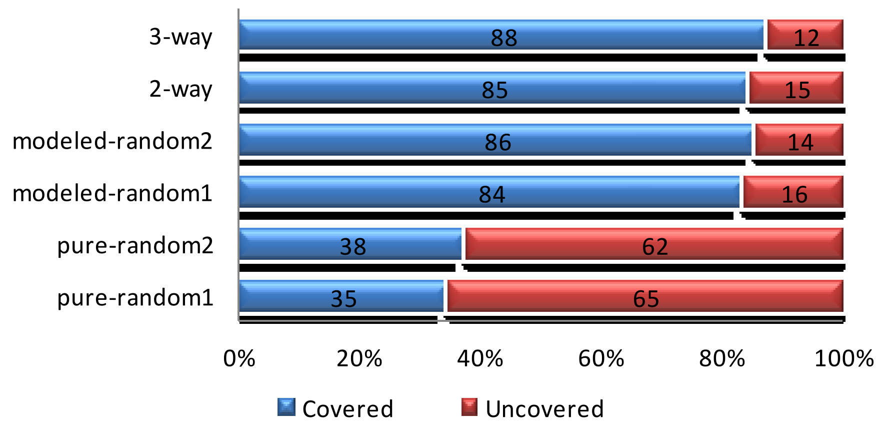


Figure 4.10 ACTS code coverage results

The results show that pure-random testing was not able to perfume well. To perform modeled-random testing we used the same model but we generated the test cases randomly.

Table 4.25 ACTS fault detection results

| Subject Programs | ACTS | |
|---|---|---|
| | Killed | not killed |
| pure-random1 | 0 | 1 |
| pure-random2 | 0 | 1 |
| modeled-random1 | 1 | 0 |
| modeled-random2 | 1 | 0 |
| 2-way | 1 | 0 |
| 3-way | 1 | 0 |

The results show that fault detection of modeled-random and t-way testing for this subject is the same. T-way testing gets the better coverage than modeled-random.

The following are some examples of the detected faults. The red lines in Figure 4.5 show the test cases that detect bugs. For example, the first line is a bug with the scenario that system let the user enter a space character, which is an invalid value for the Enum type. The second red line is another bug with the scenario that the system let the user select an invalid range for the Range types. Both of these bugs are detected during robustness testing of GUI. The red line in Figure 4.6 also shows another detected bug with the scenario that the system lets the user change the name of the parameter that is involved in the constraints.

The following are two examples of the detected faults of the sequences that lead us to find the fault in ACTS. Assume L is our node list. L1 and L2 are two different test sequences that led us to detect three different bugs in ACTS.

- *L1 = {open, import, build (extend mode), save, close}*

In this scenario, the user opens a system, imports the test set and builds it. An error was detected when we built the system in this scenario. The imported test set had an invalid format, which caused the build process to throw an exception. This error was not detected by functional testing of the import use-case individually. The import method failed to correctly set all the values that are needed by the back-end system parameters. However, this problem was not observed from outside when we tested the import use-case. The build operation after the import operation helped to expose the incorrect state as an exception that can be observed from outside.

- *L2 = {open, build, Edit a parameter, build (extend mode)}*

In this scenario, the user opens a system, builds the system, and edits a parameter. The Edit operation, as explained in “modify parameter” section, allows values of a parameter to be added or removed. After modifying a parameter, the user builds the system again. An error was detected after we called the build method again, this time in the extend mode. This error was not detected by testing the “modify parameter” use-case individually. Similar to L1, the modify method failed to correctly set all of the values to the back-end system parameters. This problem was, however, only be exposed when we built the system again.

### *4.5.2 Ant*

*Apache Ant* from SIR website [20], consists of 80500 lines of java code, and performs as a Java-based build tool. The program reads a XML-based configuration file where describes the build process and its dependencies. Ant has 6 associated versions with a single seeded fault.

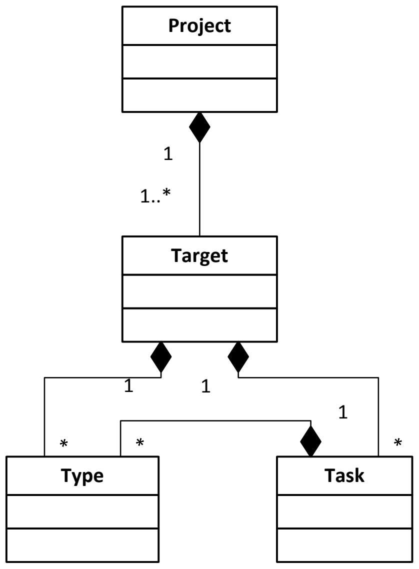


Figure 4.11 Apache Ant buildfile structure model

#### 4.5.2.1 Modeling for the Buildfile

Apache Ant's buildfiles are written in XML. Each buildfile consists of one project and at least one (default) target. Targets contain task and type elements. Figure 4.11 represents the structure of the Apache Ant's buildfile as a graph model.

##### 4.5.2.1.1 Project

A project has three attributes; name, default (a target to start with), and basedir. For the testing purposes, the base directory is always the address of the test directory.

In addition, as it is shown in the Ant graph model, each project defines one or more targets. The number_of_targets parameter with values [one, two or more] is introduced to the project model (Table 4.26) to cover the number of targets a project can define. When starting

Ant, we can select which target(s) we want to have executed. The project's default target is used when no target is given. So, the default attribute will be ignored if a target is selected. Therefore, the target_selected parameter is introduced.

Targets can depend on other targets. For example you might have a target for compiling and a target for creating a distributable. You can only build a distributable when you have compiled first, so the distribute target depends on the compile target.

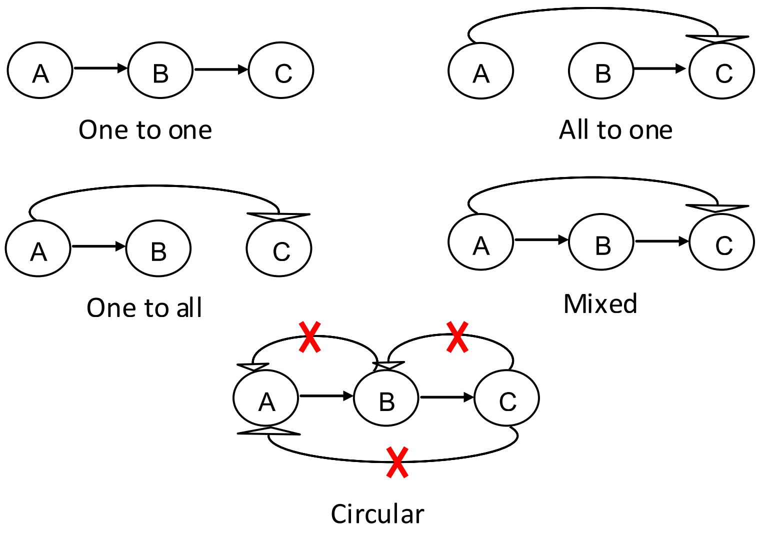


Figure 4.12 Example of dependency chains

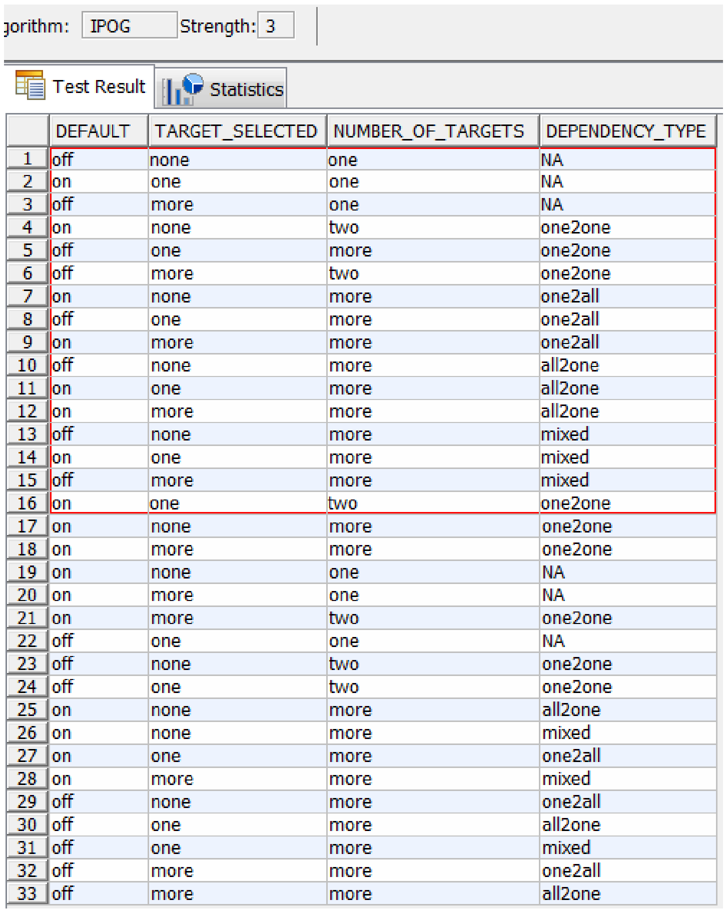
The *target_dependency_type* parameter is introduced to cover all different kinds of dependency chains that it is allowed between the targets. If we only have one target, then the value of *target_dependency_type* parameter is 'NA'. Since circular dependencies is not allowed, If we have two targets only one of them can be depend on the other; therefore, the value of *target_dependency_type* parameter is 'one to one'. For three or more targets we can have other dependencies such as 'one to all', 'all to one' and 'mixed'. As a result, following constraints are introduced between the two parameters and their values:

```
Number_of_targets='one' => target_dependency_type='NA'
Number_of_targets='two'=>target_dependency_type='none'|| target_dependency_type='one to one'
Number_of_targets='three or more' => target_dependency_type!='NA'
```

Table 4.26 Test factors of project

| Test Factor | Test Value |
|---|---|
| Default | [on, off] |
| Target_selected | [none,one,more] |
| Numbe_of_targets | [one, two or more] |
| Target_dependency_type | [NA, none, one to one, one to all, all to one, mixed] |

Figure 4.12 shows the dependencies with an example which A, B and C are different targets. The test factors (input model) of the project are shown in Table 4.26.



| | DEFAULT | TARGET_SELECTED | NUMBER_OF_TARGETS | DEPENDENCY_TYPE |
|---|---|---|---|---|
| 1 | off | none | one | NA |
| 2 | on | one | one | NA |
| 3 | off | more | one | NA |
| 4 | on | none | two | one2one |
| 5 | off | one | more | one2one |
| 6 | off | more | two | one2one |
| 7 | on | none | more | one2all |
| 8 | off | one | more | one2all |
| 9 | on | more | more | one2all |
| 10 | off | none | more | all2one |
| 11 | on | one | more | all2one |
| 12 | on | more | more | all2one |
| 13 | off | none | more | mixed |
| 14 | on | one | more | mixed |
| 15 | off | more | more | mixed |
| 16 | on | one | two | one2one |
| 17 | on | none | more | one2one |
| 18 | on | more | more | one2one |
| 19 | on | none | one | NA |
| 20 | on | more | one | NA |
| 21 | on | more | two | one2one |
| 22 | off | one | one | NA |
| 23 | off | none | two | one2one |
| 24 | off | one | two | one2one |
| 25 | on | none | more | all2one |
| 26 | on | none | more | mixed |
| 27 | on | one | more | one2all |
| 28 | on | more | more | mixed |
| 29 | off | none | more | one2all |
| 30 | off | one | more | all2one |
| 31 | off | one | more | mixed |
| 32 | off | more | more | one2all |
| 33 | off | more | more | all2one |

Figure 4.13 Project test cases

#### 4.5.2.1.2 Target

A target has five attributes; name, depends, if, unless, and description. As it is explained in previous section, targets can depend on other targets. The value of depends attribute is a comma-separated list of the names of targets to which this target depends and it is defined based on the value of number_of_targets and target_dependency_type. It is basically related to multiple targets and not one individual target.

A target name can be any alphanumeric string valid in the encoding of the XML file. The empty string "", comma ",", and space " " are included in this set. In this study, we only used valid test cases, i.e., invalid tests were discarded.

The optional description attribute can be used to provide a one-line description of this target, which is printed by the -projecthelp command-line option. Targets without such a description are deemed internal and will not be listed, unless either the -verbose or -debug option is used. Therefore, in order to test these options, we need to have buildfiles with and without descriptions. But description attribute itself will not change the output behavior of a target.

A target also has the ability to perform its execution if (or unless) a property has been set. This allows, for example, better control on the building process depending on the state of the system (java version, OS, command-line property defines, etc.). To make a target sense this property, we should add the if (or unless) attribute with the name of the property that the target should react to .The if and unless attributes only enable or disable the individual target to which they are attached.

In addition a target is a container of tasks that cooperate to reach a desired state during the build process. A target contains one or more tasks. There is a set of 17 groups of built-in tasks available in Ant, but it is easy to write your own. In this study we are only focused on the built-in tasks. Given the large number of tasks available with Ant, it may be difficult to get an overall view of what each task can do. Therefore, the tasks are further categorized to 17 groups such as archive, document, compile, file, logging, execution, and etc.

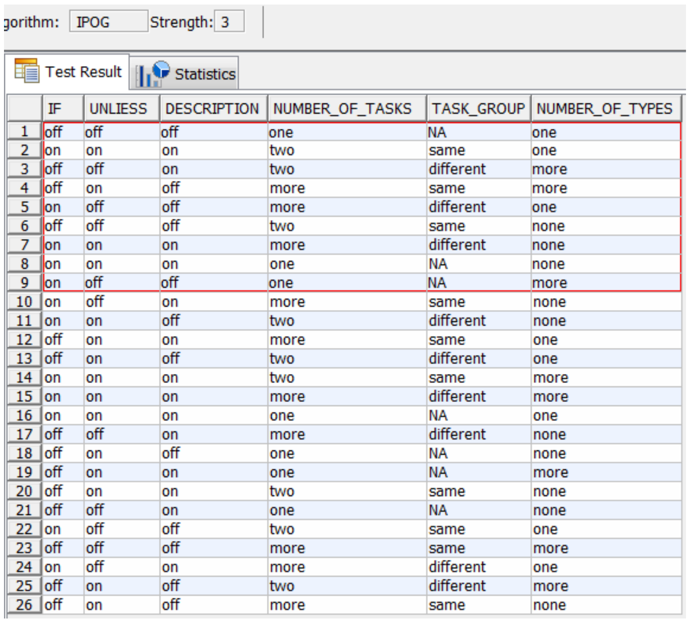
The number_of_task parameter is identified to cover one task or more in each target. Also when we have two or more tasks, we would like to cover the cases that all tasks are from same group or different groups. The test factors of the individual target are shown in Table 4.27.

A sample concrete test case contains a target with two different task groups (compile and file tasks) is:

```
<target name="build" >
        <mkdir dir="build"> //file task
        <javac srcdir="build" destdir="build"> //compile task
              <classpath refid="classpath" />
              <exclude name="*.java" />
        </javac>
</target>
```

Table 4.27 Test factors of target

| Test Factor | Test Value |
|---|---|
| If | [on, off] |
| Unless | [on,off] |
| Description | [on,off] |
| number_of_tasks | [one, two, more] |
| Task_group | [NA, same_group, different_group] |
| Number_of_types | [none, one, more] |


gorithm: IPOG Strength: 3

| | IF | UNLIESS | DESCRIPTION | NUMBER_OF_TASKS | TASK_GROUP | NUMBER_OF_TYPES |
|---|---|---|---|---|---|---|
| 1 | off | off | off | one | NA | one |
| 2 | on | on | on | two | same | one |
| 3 | off | off | on | two | different | more |
| 4 | off | on | off | more | same | more |
| 5 | on | off | off | more | different | one |
| 6 | off | off | off | two | same | none |
| 7 | on | on | on | more | different | none |
| 8 | on | on | on | one | NA | none |
| 9 | on | off | off | one | NA | more |
| 10 | on | off | on | more | same | none |
| 11 | on | on | off | two | different | none |
| 12 | off | on | on | more | same | one |
| 13 | off | on | off | two | different | one |
| 14 | on | on | on | two | same | more |
| 15 | on | on | on | more | different | more |
| 16 | on | on | on | one | NA | one |
| 17 | off | off | on | more | different | none |
| 18 | off | on | off | one | NA | none |
| 19 | off | on | on | one | NA | more |
| 20 | off | on | on | two | same | none |
| 21 | off | off | on | one | NA | none |
| 22 | on | off | off | two | same | one |
| 23 | off | off | off | more | same | more |
| 24 | on | off | on | more | different | one |
| 25 | off | on | off | two | different | more |
| 26 | off | on | off | more | same | none |

Figure 4.14 Target test cases

#### 4.5.2.1.3 Task

A task is a piece of code that can be executed and has multiple attributes. Task structure includes <***name*** attribute1="value1" attribute2="value2"…/> where ***name*** is the name of the task, attributeN is the attribute name, and valueN is the value of this attribute. All tasks share a task *name* attribute and *id* attribute. The value of *name* attribute will be used in the logging messages generated by Ant. Also task *name* and *id* are both unique identifies.

As we discussed in previous section, there are 17 different built-in task groups available in Ant. We will model each task group separately. In the Ant specification, a table of parameters is available for each task with a description column for each attribute. We used this table to identify the test factors and their values as well as relations and constraints between them. Some of these parameters are not interesting from testing perspective; therefore, we will omit them from set of test factors. In addition, we need to identify all behaviors, features, and environmental parameters that may affect the behavior of the system.

4.5.2.1.3.1 Archive tasks

Tasks such as zip, jar, tar, untar, unjar, unzip, and 12 more tasks are in this category. We use the zip task to explain in details and the model for other tasks is similar to this.

4.5.2.1.3.1.1 Zip task

This task will create a zipfile. The test factors of zip are shown in Table 4.28. In this table, the first column calls zip attributes. Note that not all the attributes of an element are always used as the test factors and also sometimes it is needed to add extra parameters.

The zipfile is the old name for the destfile. It is the zip file that after execution of this task will be created. Also the name of this file is not important as the testing perspective and our concrete test case generator will automatically assign a value to this attribute. As the specification is stated having one of these two parameters is required and they shouldn't be called at the same time; therefore, we need to introduce some constraints:

Destfile='off' => zipfile='on'
Destfile='on' => zipfile='off'

The zip task form an implicit FileSet (FileSet is a Type which we will talk about it later in type section) which means it supports most attributes of <fileset> such as *include* and *exclude*. In addition we can use nested <fileset> as well.

The *basedir* is the address of the directory from which zip the files and it can be an empty directory. This attribute is optional and the name of the directory will not change the behavior of the system unless it is a wrong address. But the implicit file set is only used if the *basedir* is set.

The *Includesfile*/*excludesfile* attribute specifies the name of a file which each line of this file is include/exclude pattern respectively. Also includes and excludes attributes both are comma- or space-separated list of patterns that must be included or excluded. So, basically we have three ways to specify the pattern of files that must be included or excluded, using a list, a file, or a nested fileset. Two test factors are identified: include and exclude, with four values [list, file, nested, off]. If the value of this parameter sets to 'file' in a test case then a predefined file contains a list of patterns e.g. *.java *.java , *.jar will be assign to the includesfile attribute.

Moreover, we do not consider the number of patterns inside the list or the file as a separate parameter. We assume that if the program is correct while having only one pattern e.g. *.java, then it will be also correct when there are more patterns.

We identified a relation between four parameters, *compress, keepcompression,, update,* and *Destfileexist.* The c*ompress* indicates that not only it stores the data but also compresses them. If we update an existing archive not only the files we've added recently but also the entire archive will be updated. If we set the *keepcompression* to false then the compression will not be applied to the new files and the compression of older files will not be changed. The default value of *keepcompression* is set to 'false'. The value of *keepcompression* parameter will change the behavior of the compress if the *compress* value is set to 'true'. Also we will not see any difference in the output if we are not updating an existing archive that was not compressed before. So, not only we need to have an existing archive file, but also the existing file should not be getting compressed originally. *Update* parameter indicates whether to update or overwrite the destination file if it already exists. *Destfileexist* parameter is identified as an environmental variable. If the destination file does not exist then the value is 'none'.

Table 4.28 Zip task test factors

| Test factors | Test values |
|---|---|
| Destfile | [on, off] |
| Zipfile | [on, off] |
| Basedir | [on, off, empty] |
| Include | [list, file, nested, off] |
| Exclude | [list, file, nested, off] |
| nestedfileset | [NA, one, two or more] |
| Compress | [true,false] |
| Keepcompression | [true,false] |
| Update | [off,update,overwrite] |
| Destfileexist | [none, compressed, duplicate ] |
| Whenempty | [fail, skip, create] |
| Duplicate | [add, preserve, fail] |

The parameters *whenempty* and *duplicate,* represent the behavior of the zip task when no file or duplicate file is found. Therefore, in the environment we need to have both scenarios.

For example, the scenario number 1 happens when the *basedir* is empty; therefore, no file is found for the *whenempty* parameter. We have already covered this scenario in the ‘empty’ value of basedir. The scenario number 2 happens when we have an existing archive contains ‘file1’, and we are going to update it with the same ‘file1’; therefore, the duplicate file is found, which we have covered this one using the ‘duplicate’ value of *Destfileexist* parameter. For example if we have an abstract test case:

| | |
|---|---|
| destfile = on | Compress = false |
| zipfile = off | Keepcompression = false |
| basedir = on | Update = update |
| Include = list | Destfileexist = duplicate |
| Exclude = off | Whenempty = skip |
| Nestedfileset=NA | Duplicate = add |

The automatic test case generator will read this test case and it will generate a <zip> tag:

<zip destfile=”test.zip” basedir=”.” Includes=”*.java” compress=”false” update=”true” whenempty=”skip” duplicate=”add” />

Also it will prepare the test environment with a zip file and a java file in the test directory, that this zip file contains the same java file.

A loop exists between the nested elements of Type elements. Since the Apache Ant is a large system, in the following paragraphs, we would like to focus on the part of the buildfile that contains loop between their elements and discuss our modeling process for it.

#### 4.5.2.1.4 Type

Ant supports different types such as FileSet, Selectors, Mapper, and etc. For example, a FileSet is a group of files. These files can be found in a directory tree starting in a base directory and are matched by user defined patterns. While in Ant the source files are usually specified as filesets, we don't specify target files directly. But we can tell the Ant how to find the target file for one source file. The Mapper type is responsible for this. It constructs target file names based on user-defined rules.

As we discussed in section 3.1.2 and as it is shown in Figure 3.3, there exists a loop involving nodes, ‘classpath’, ‘fileset’, ‘depend’, and ‘mapper’. We break the loop by removing the back edge going from node ‘classpath’ to ‘fileset’. After we modeled the graph without loop then we add the back edge and remodel the ‘classpath’ node. The ‘classpath’ node using the ‘files et’ node as its nested element and we have already modeled the ‘fileset’ node which we can add to the model of ‘classpath’.

4.5.2.1.4.1 FileSet

A FileSet is a group of files. These files can be found in a directory tree starting in a base directory and are matched by user defined patterns. The root of the directory tree will specify with the ‘dir’ attribute. If we only have a single-file then instead of specifying the ‘dir’ we need to specify the file with the ‘file’ attribute. Therefore, the two identified parameters ‘dir’ and ‘file’ in Table 4.29 cannot be active at the same time. The constraints are needed:

dir=’on’ => file=’off’
file=’on’ => dir=’off’

Table 4.29 Fileset test factors

| Test Factors | Test values |
|---|---|
| Dir | [on, off] |
| File | [on, off] |
| Defaultexcludes | [on, off] |
| Includes | [list, file, nested, off] |
| Excludes | [list, file, nested, off] |
| Casesensitive | [on, off] |
| Nested_Depend | [on, off] |

The values for ‘include’ and ‘exclude’ parameters are identified as we discussed in pervious paragraphs. If the value of ‘casesensitive’ parameter is ‘on’ then include/exclude patterns must be treated in a case sensitive way. In addition, FileSet has <depend> as its nested element.

#### 4.5.2.1.4.2 Depend

The <depend> selects a file whose last modified date is later than another equivalent file in another location. The <depend> tag supports the use of a contained <mapper> element to define the location of the file to be compared against. The precise location depends on a combination of this attribute and the <mapper> element, if any. If no <mapper> element is specified, the identity type mapper is used. This element only takes the two attributes 'targetdit' and 'granularity'. The 'targetdit' is not an optional attribute. The granularity' is the number of milliseconds leeway to give before deciding a file is out of date. This is needed because not every file system supports tracking the last modified time to the millisecond level. Default is 0 milliseconds.

Table 4.30 Depend Type test factors

| Test Factors | Test values |
|---|---|
| Targetdir | [on,off] |
| Granularity | [off, 0, 5] |
| Nested_Mapper | [on, off] |

#### 4.5.2.1.4.3 Mapper

We can tell Ant how to find the target file(s) for one source file. Mapper is responsible for this. It constructs target file names based on rules that can be parameterized with 'from' and 'to' attributes. The <mapper> element has the following attributes:

Table 4.31 Mapper test factors

| Test Factors | Test values |
|---|---|
| Type | [identity, flatten, merge, glob, package, unpackage, regexp, off] |
| Classname | [on, off] |
| Classpath | [on, off, nested] |
| From | [on, off] |
| To | [on, off] |

The 'type' parameter will specify one of the built-in implementation of Mapper. We can specify this implementation by 'classname' also. Therefore, if we specify the 'type' then the 'classname' should be 'off'. If the 'classname' is 'off' then the 'classpath' should be 'off' too because the ‘classpath’ is used to look up the ‘classname’. The class path can be specified via a nested <classpath> as well. Therefore, the value ‘nested’ is added to the test values of ‘classpath’ parameter.

4.5.2.1.4.4 Classpath

The <classpath> will specify the ‘location’ or ‘path’ of the class. The ‘location’ attribute specifies a single file or directory relative to the project's base directory (or an absolute filename), while the path attribute accepts colon- or semicolon-separated lists of locations. The <classpath> element has the <fileset> element as its nested attribute. Since we have already modeled the <fileset> it is convenient to recall it again in this element as well.

Table 4.32 Classpath test factors

| Test Factors | Test values |
|---|---|
| Path | [on, off] |
| Location | [on, off] |
| Nested_fileset | [on, off] |

Following example is a concrete <classpath> for the abstract test case:

Table 4.33 Example of concrete test case for the <classpath>

| | Abstract test cases | Concrete value |
|---|---|---|
| Classpath | path=’off’<br>Location=’off’<br>Nested_fileset=’on’ | <classpath><br>    <fileset dir=”lib”><br>        <include name=”*.class”/><br>    </fileset><br></classpath> |
| Fileset | dir=’on’<br>Includes=’nested’<br>…..<br>…… | |

In this example since the value of ‘nested_fileset’ is ‘on’ therefore a nested fileset element will be selected from the <fileset> abstract test cases.

4.5.2.2 Modeling for the Command Line Interface

The various options are available in command line interface as shown in Table 4.17. Some of these options e.g. projecthelp, version, diagnostics are totally independent from each other. Because there is no interaction between them they must appear in the test only once.

Table 4.34 Command line model

| Test Factors | Test Values | Description |
|---|---|---|
| projecthelp | [on, off] | print project help information |
| version | [on, off] | print the version information and exit |
| Diagnostics | [on, off] | print information to diagnose or report problems |
| Quite | [on, off] | be extra quiet |
| Verbose | [on, off] | be extra verbose |
| Debug | [on, off] | print debugging information |
| lib <path> | [on, off] | specifies a path to search for jars and classes |
| logfile <file> | [on, off] | use given file for log |
| logger <classname> | [on, off] | the class which is to perform logging |
| listener <classname> | [on, off] | add an instance of class as a project listener |
| Noinput | [on, off] | do not allow interactive input |
| buildfile <file> | [on, off] | use given buildfile |
| keep-going | [on, off] | execute all targets that do not depend on failed target |
| inputhandler <class> | [on, off] | the class which will handle input requests |
| find <file> | [on, off] | search for buildfile towards the root |
| Noclasspath | [on, off] | Run ant without using CLASSPATH |
| Nouserlib | [on, off] | Run ant without using the jar files library |
| Autoproxy | [on, off] | Java 1.5+ : use the OS proxies |

Some other options e.g. debug, quite, and verbose affect the amount of logging output by Ant.

Table 4.35: Summary of Ant input model

| Ant | # of parameter values | # of parameters | # of constraints and the involved parameters | # of constraints | # of relations and the number of involved parameters | # of relations |
|---|---|---|---|---|---|---|
| Jar/tar/war/zip | $2^4 3^5 4^2$ | 11 | $2^3$ | 3 | $4^1$ | 1 |
| Unzip/unwar/unjar | $2^3$ | 3 | $2^2$ | 2 | - | - |
| gzip/bzip2 | $2^3$ | 3 | $2^2$ | 2 | - | - |
| untar | $2^3 3^1$ | 4 | $2^2$ | 2 | - | - |
| Include/exclude | $2^3$ | 3 | - | - | - | - |
| contains | $2^7 3^1$ | 8 | $6^7$ | 7 | $2^1 3^1$ | 2 |
| depend | $2^2 3^1$ | 3 | $1^1$ | 1 | - | - |
| Date | $2^3 3^3$ | 6 | $1^1 2^2$ | 3 | $2^2$ | 2 |
| or/and/not | $2^2$ | 2 | - | - | - | - |
| different | $2^2 3^1$ | 3 | - | - | - | - |
| Size | $3^2 6^1$ | 3 | - | - | $2^2$ | 2 |
| selector | $2^2 3^6$ | 8 | $2^2$ | 2 | - | - |
| fileset | $2^5 4^2$ | 7 | $2^2$ | 2 | - | - |
| patternset | $4^2$ | 2 | - | - | - | - |
| Taglet/doclet | $2^2$ | 2 | - | - | - | - |
| package | $2^2$ | 2 | - | - | - | - |
| Link | $2^3$ | 3 | $3^1$ | 1 | - | - |
| doctitle | $2^2$ | 2 | - | - | - | - |
| param | $2^2$ | 2 | - | - | - | - |
| group | $2^3$ | 3 | - | - | - | - |
| mapper | $2^3 3^1 8^1$ | 5 | $2^2$ | 2 | - | - |
| classpath | $2^3$ | 3 | - | - | - | - |
| javac | $2^7 4^2 8^1$ | 10 | $2^2$ | 2 | $2^1$ | 1 |
| bottom | $2^2$ | 2 | - | - | - | - |
| Ant | $2^6$ | 6 | - | - | $2^1$ | 1 |
| antcall | $2^4$ | 4 | $1^1$ | 1 | - | - |
| Java | $2^6 3^1$ | 7 | $2^2$ | 2 | - | - |
| reference | $2^5 4^2$ | 7 | - | - | - | - |
| property | $2^8 3^1$ | 9 | $3^1 5^1$ | 2 | - | - |
| checksum | $2^4 3^1 4^2$ | 7 | - | - | $2^1$ | 1 |
| Copy | $2^5 3^1$ | 6 | $3^2$ | 2 | - | - |
| concat | $2^4$ | 4 | - | - | - | - |
| replace | $2^5 4^2$ | 7 | $2^3$ | 3 | - | - |
| record | $2^3 3^1 6^1$ | 5 | $5^1$ | 1 | - | - |
| Target | $2^3 3^3$ | 6 | $2^2$ | 2 | - | - |
| project | $2^1 3^2 6^1$ | 4 | $2^3$ | 3 | - | - |
| TOTAL | $2^{122} 3^{31} 4^{14} 6^3 8^2$ | 172 | $1^3 2^{29} 3^4 5^2 6^7$ | 45 | $2^8 3^1 4^1$ | 10 |

When no arguments are specified, Ant looks for a build.xml file in the current directory and, if found, uses that file as the build file and runs the target specified in the default attribute of the <project> tag. To make Ant use a build file other than build.xml, use the command-line option -buildfile file, where file is the name of the build file you want to use.

4.5.2.3 Results and discussion

The designed model for Ant has 53 IPMs which are shown in Table 4.35, yielding 2121 generated concrete test cases. The relation between abstract and concrete test cases is one to many; therefore, we were able to cover all the test cases.

Code coverage data are shown in Figure 4.15. We used *Clover* to collect code coverage. Ant statistics are shown in Table 4.36. e.g. number of lines of code in Ant are 80500.

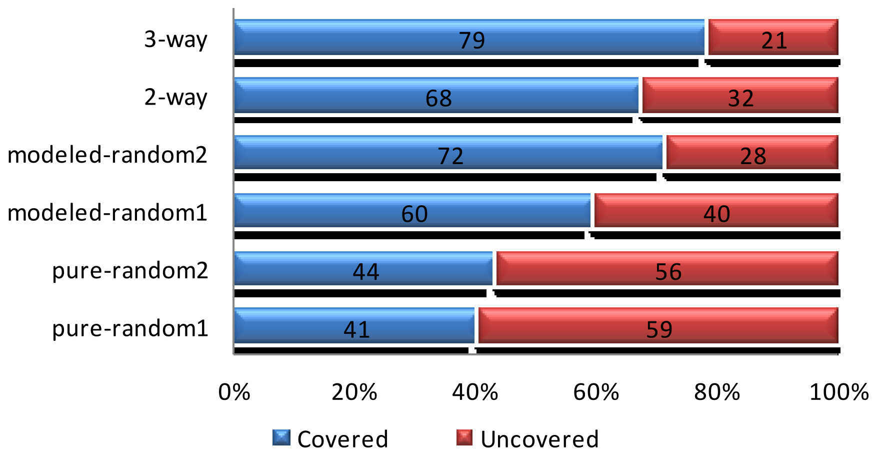


Figure 4.15 Ant code coverage results

Cover gave us the code coverage for all of the test cases. While we executed our tests, it highlighted the parts of the source code that were executed. It is shown in that our tests covered almost 80% of lines.

Table 4.36 Ant statistics

| | |
|---|---|
| LOC (line of code) | 80,500 |
| Number of Statements | 38,271 |
| Number of Branches | 16,576 |
| Number of Methods | 7,434 |
| Number of Classes | 627 |
| Number of Packages | 67 |
| Number of faulty versions | 6 |
| Type of faults | Seeded |

The total number of detected faults is 5 out of 6.

Table 4.37 Ant fault detection results

| Subject Programs | Space | |
|---|---|---|
| | Killed | not killed |
| pure-random1 | 1 | 5 |
| pure-random2 | 1 | 5 |
| modeled-random1 | 3 | 3 |
| modeled-random2 | 4 | 2 |
| 2-way | 4 | 2 |
| 3-way | 5 | 1 |

### *4.5.3 Space*

*Space* from SIR website [20], consists of 9564 lines of C code, and performs as an interpreter for an array definition language (ADL). The program reads a file that contains several ADL statements, and checks the contents of the file for adherence to the ADL grammar and to specific consistency rules. If the ADL file is correct, space outputs an array data file containing a list of array elements, positions, and excitations; otherwise the program outputs error messages.

Space has 35 associated versions, each containing a single fault that had been discovered during the program's development.

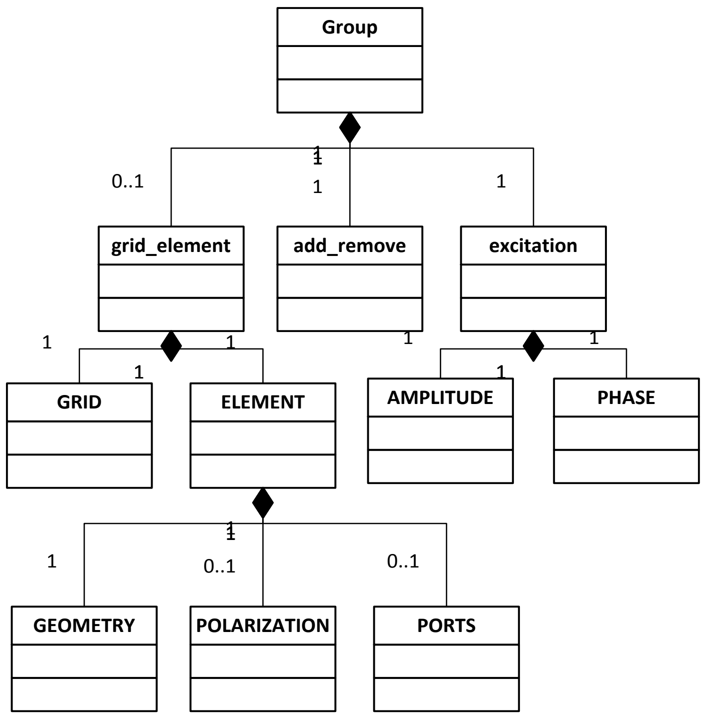


Figure 4.16 Space ADL file structure model

#### 4.5.3.1 Modeling for the ADL file

The ADL file contains of three main elements; grid and element definition, add and remove definitions and group excitation. The first element is an optional element but the other two are required.

The grid_element contains two parts. The first part related to GRID definition and the second part is the ELEMENT definitions and both of them are required. Also ELEMENT contains two optional parts: PORTS and POLARIZATION, and on required part: GEOMETRY.

In addition, the excitation element also contains of two required parts: PHASE and AMPLITUDE. Figure 4.16 represents the structure of the ADL file as a graph model.

#### 4.5.3.1.1 Grid

The program contains four types of grid definition: square, rectangular, hexangular and triangular. A grid is defined by its width and length.

A square and hexangular grid is defined in a single_step definition, because we only need to have the value of one length.

A rectangular needs a double_step definition, because we not only need to have the length but also width. The length units are millimeter, centimeter and meter.

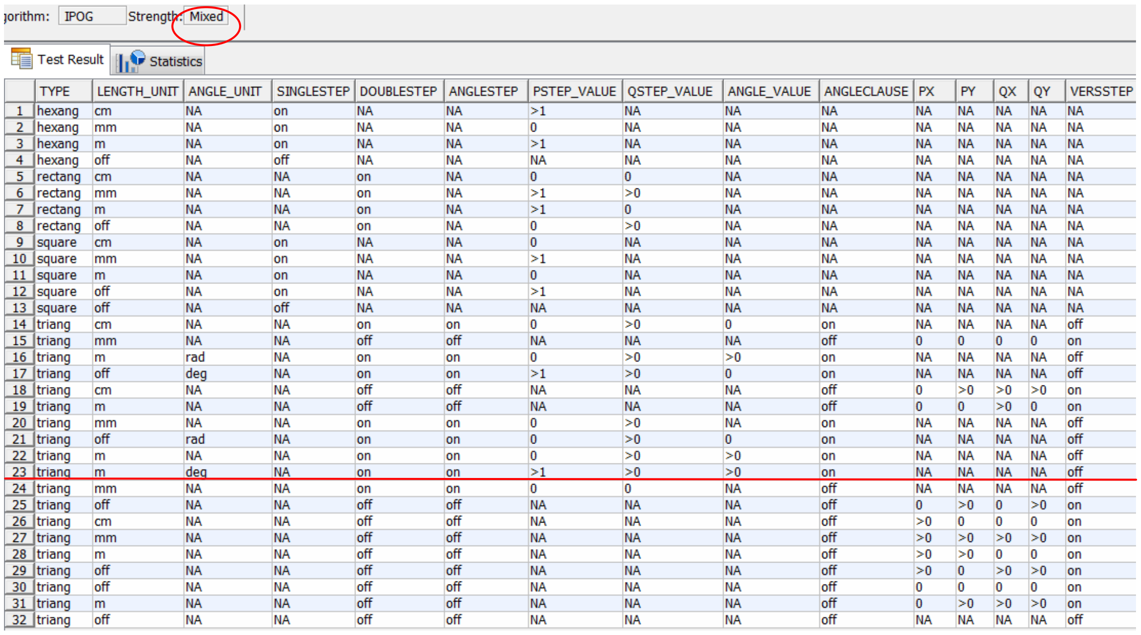
A triangular grid is defined by its angle, width and length, which is called as angle_step definition. The angle unit units are degree and radiant.

Table 4.38 Test factors of grid

| Test factors | Test values |
|---|---|
| grid_type | [square, hexang, rectang, triang] |
| length_unit | [off, mm, cm, m] |
| angle_unit | [NA, rad, deg] |
| Singlestep | [NA, on, off] |
| Doublestep | [NA, on, off] |
| Anglestep | [NA, on, off] |
| Verssetp | [NA, on, off] |
| Pstep_value | [NA, 0, >0] |
| Qstep_value | [NA, 0, >0] |
| Angle_value | [NA, 0, >0] |
| angleclause | [NA, on, off] |
| PX | [NA, 0, >0] |
| PY | [NA, 0, >0] |
| QX | [NA, 0, >0] |
| QY | [NA, 0, >0] |

Now we need to identify the relation and constraint between test factors. If the grid type is not triangular, then all the test factors related to angle should assign to 'NA'. Also if the grid type is not rectangular then the single step should also assign to ‘NA’. Following constraints will be added to the system.

```
type="square" || type="hexang" => doublestep="NA" && anglestep="NA" &&
singlestep!="NA" && angleclause="NA" && versstep="NA"
type="rectang" => singlestep="NA" && pstep_value!="NA" && qstep_value!="NA" &&
anglestep="NA" && versstep="NA"
type="triang" => singlestep="NA" && anglestep!="NA" && versstep!="NA" &&
doublestep!="NA"
singlestep="on" => qstep_value="NA" && pstep_value!="NA"
singlestep="off" => pstep_value="NA"&& length_unit="off"
doublestep!="on" => qstep_value="NA"
doublestep!="on" && singlestep!="on" && versstep!="on" => length_unit="off"
doublestep="on" => pstep_value!="NA" && qstep_value!="NA"
anglestep="NA" => angleclause="NA"
anglestep="on" => angleclause!="NA" && versstep="off" && doublestep="on"
anglestep="off" => doublestep="off" && angleclause="off" && angle_value="NA" &&
angle_unit="NA" && pstep_value="NA" && qstep_value="NA"
angleclause!="on" => angle_value="NA" && angle_unit="NA"
angle_value="NA" => angle_unit="NA"
versstep="on" => anglestep="off" && px!="NA" && py!="NA" && qx!="NA" && qy!="NA"
versstep!="on" => px="NA" && py="NA" && qx="NA" && qy="NA"
```



jorithm: IPOG Strength: Mixed

Test Result | Statistics

| | TYPE | LENGTH_UNIT | ANGLE_UNIT | SINGLESTEP | DOUBLESTEP | ANGLESTEP | PSTEP_VALUE | QSTEP_VALUE | ANGLE_VALUE | ANGLECLAUSE | PX | PY | QX | QY | VERSSTEP |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | hexang | cm | NA | on | NA | NA | >1 | NA | NA | NA | NA | NA | NA | NA | NA |
| 2 | hexang | mm | NA | on | NA | NA | 0 | NA | NA | NA | NA | NA | NA | NA | NA |
| 3 | hexang | m | NA | on | NA | NA | >1 | NA | NA | NA | NA | NA | NA | NA | NA |
| 4 | hexang | off | NA | off | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |
| 5 | rectang | cm | NA | NA | on | NA | 0 | 0 | NA | NA | NA | NA | NA | NA | NA |
| 6 | rectang | mm | NA | NA | on | NA | >1 | >0 | NA | NA | NA | NA | NA | NA | NA |
| 7 | rectang | m | NA | NA | on | NA | >1 | 0 | NA | NA | NA | NA | NA | NA | NA |
| 8 | rectang | off | NA | NA | on | NA | 0 | >0 | NA | NA | NA | NA | NA | NA | NA |
| 9 | square | cm | NA | on | NA | NA | 0 | NA | NA | NA | NA | NA | NA | NA | NA |
| 10 | square | mm | NA | on | NA | NA | >1 | NA | NA | NA | NA | NA | NA | NA | NA |
| 11 | square | m | NA | on | NA | NA | 0 | NA | NA | NA | NA | NA | NA | NA | NA |
| 12 | square | off | NA | on | NA | NA | >1 | NA | NA | NA | NA | NA | NA | NA | NA |
| 13 | square | off | NA | off | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA | NA |
| 14 | triang | cm | NA | NA | on | on | 0 | >0 | 0 | on | NA | NA | NA | NA | off |
| 15 | triang | mm | NA | NA | off | off | NA | NA | NA | off | 0 | 0 | 0 | 0 | on |
| 16 | triang | m | rad | NA | on | on | 0 | >0 | >0 | on | NA | NA | NA | NA | off |
| 17 | triang | off | deg | NA | on | on | >1 | >0 | 0 | on | NA | NA | NA | NA | off |
| 18 | triang | cm | NA | NA | off | off | NA | NA | NA | off | 0 | >0 | >0 | >0 | on |
| 19 | triang | m | NA | NA | off | off | NA | NA | NA | off | 0 | 0 | >0 | 0 | on |
| 20 | triang | mm | NA | NA | on | on | 0 | >0 | NA | on | NA | NA | NA | NA | off |
| 21 | triang | off | rad | NA | on | on | 0 | >0 | 0 | on | NA | NA | NA | NA | off |
| 22 | triang | m | NA | NA | on | on | 0 | >0 | >0 | on | NA | NA | NA | NA | off |
| 23 | triang | m | deg | NA | on | on | >1 | >0 | >0 | on | NA | NA | NA | NA | off |
| 24 | triang | mm | NA | NA | on | on | 0 | 0 | NA | off | NA | NA | NA | NA | off |
| 25 | triang | off | NA | NA | off | off | NA | NA | NA | off | 0 | >0 | 0 | >0 | on |
| 26 | triang | cm | NA | NA | off | off | NA | NA | NA | off | >0 | 0 | 0 | 0 | on |
| 27 | triang | mm | NA | NA | off | off | NA | NA | NA | off | >0 | >0 | >0 | >0 | on |
| 28 | triang | m | NA | NA | off | off | NA | NA | NA | off | >0 | >0 | 0 | 0 | on |
| 29 | triang | off | NA | NA | off | off | NA | NA | NA | off | >0 | 0 | >0 | >0 | on |
| 30 | triang | off | NA | NA | off | off | NA | NA | NA | off | 0 | 0 | 0 | 0 | on |
| 31 | triang | m | NA | NA | off | off | NA | NA | NA | off | 0 | >0 | >0 | >0 | on |
| 32 | triang | off | NA | NA | off | off | NA | NA | NA | off | NA | NA | NA | NA | off |

Figure 4.17 Test cases of grid

An example of a concrete value for the test case #23 is:

GRID TRIANGULAR ANGLE 360 deg   PSTEP 50 QSTEP 1 m

| | |
|---|---|
| grid_type = TRIANGULAR<br>length_unit = m<br>angle_unit = deg<br>Singlestep_def = NA<br>Doublestep_def = on<br>Anglestep_def = on<br>Verssetp_def = off<br>Pstep_value = >0 | Qstep_value = >0<br>Angle_value = >0<br>angleclause = on<br>PX = NA<br>PY = NA<br>QX = NA<br>QY = NA |

#### 4.5.3.1.2 Geometry

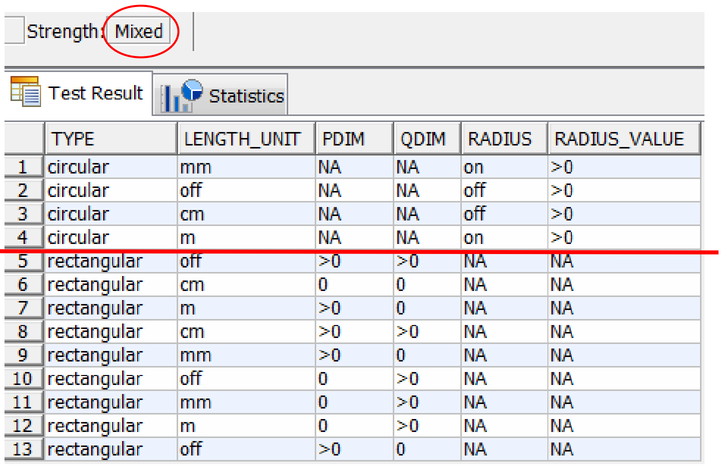
A geometry element has two types; RECTANGULAR and CIRCULAR. The rectangular geometry has a P and Q dimensions call PDIM and QDIM which is an unsigned-read value. Also the length unit is millimeter, centimeter and meter.

Table 4.39 Test factors of geometry

| Test factors | Test values |
|---|---|
| Type | [rectangular, circular] |
| length_unit | [off, mm, cm, m] |
| PDIM | [NA, 0, >0] |
| QDIM | [NA, 0, >0] |
| Radius | [NA, on, off] |
| Radius_value | [NA, >0] |

The circular geometry has radius value and unit instead of dimensions. The constraints are needed to ensure that when type is rectangular, the value of radius and radius_value are assigned to 'NA' and vice versa.

We used the mixed-relation feature of ACTS to generate the test cases as are shown in Figure 4.18. For example when the type is 'circular', we don't want to test all the different combinations between 'radius' and the 'length_unit'. In total there are only four different combinations that we would like to test for the circular type. When both 'radius' and 'length_unit' parameters are 'on', when both of them are 'off' and when one is 'off' and the other one is 'on'. In addition, we want to make sure the system is working for the different unit types (cm,mm,m) and we want to test it at least once for each unit.



| | TYPE | LENGTH_UNIT | PDIM | QDIM | RADIUS | RADIUS_VALUE |
|---|---|---|---|---|---|---|
| 1 | circular | mm | NA | NA | on | >0 |
| 2 | circular | off | NA | NA | off | >0 |
| 3 | circular | cm | NA | NA | off | >0 |
| 4 | circular | m | NA | NA | on | >0 |
| 5 | rectangular | off | >0 | >0 | NA | NA |
| 6 | rectangular | cm | 0 | 0 | NA | NA |
| 7 | rectangular | m | >0 | 0 | NA | NA |
| 8 | rectangular | cm | >0 | >0 | NA | NA |
| 9 | rectangular | mm | >0 | 0 | NA | NA |
| 10 | rectangular | off | 0 | >0 | NA | NA |
| 11 | rectangular | mm | 0 | >0 | NA | NA |
| 12 | rectangular | m | 0 | >0 | NA | NA |
| 13 | rectangular | off | >0 | 0 | NA | NA |

Figure 4.18 Test cases of geometry

The concrete value for the abstract test case is #4:

GEOMETRY CIRCULAR RADIUS 17 m

| | |
|---|---|
| type = circular<br>length_unit = m<br>PDIM = NA | QDIM = NA<br>Radius = on<br>Radius_value = >0 |

#### 4.5.3.1.3 Polarization

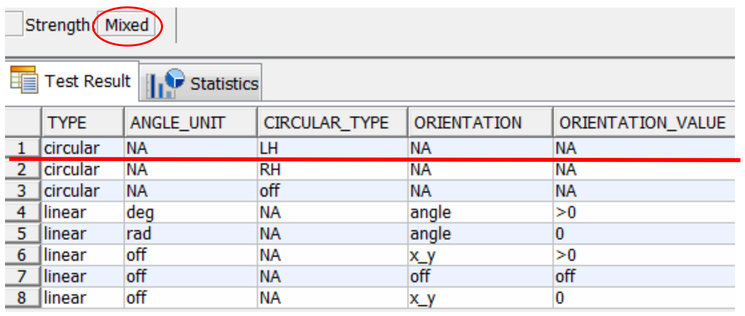
There are two kinds of polarization: LINEAR and CIRCULAR. The LINEAR polarization also can have an orientation which can be value or an angle. The Circular is either LH or RH.

Table 4.40 Test factors of polarization

| Test factors | Test values |
|---|---|
| Type | [linear, circular] |
| angle_unit | [NA, off, deg, rad] |
| Circular_type | [NA, off, LH, RH] |
| Orientation | [NA, off, x_y, angle] |
| orientation_value | [NA, >0, 0] |

The circular type only needs the circular type value and all the parameters should assign to ‘NA’. The constraints will be added accordingly.

We used the mixed-relation feature of ACTS to generate the test cases as are shown in Figure 4.19.


Strength Mixed

Test Result | Statistics

| | TYPE | ANGLE_UNIT | CIRCULAR_TYPE | ORIENTATION | ORIENTATION_VALUE |
|---|---|---|---|---|---|
| 1 | circular | NA | LH | NA | NA |
| 2 | circular | NA | RH | NA | NA |
| 3 | circular | NA | off | NA | NA |
| 4 | linear | deg | NA | angle | >0 |
| 5 | linear | rad | NA | angle | 0 |
| 6 | linear | off | NA | x_y | >0 |
| 7 | linear | off | NA | off | off |
| 8 | linear | off | NA | x_y | 0 |

Figure 4.19 Test cases of polarization

The concrete value for the abstract test case #1:

POLARIZATION CIRCULAR LH

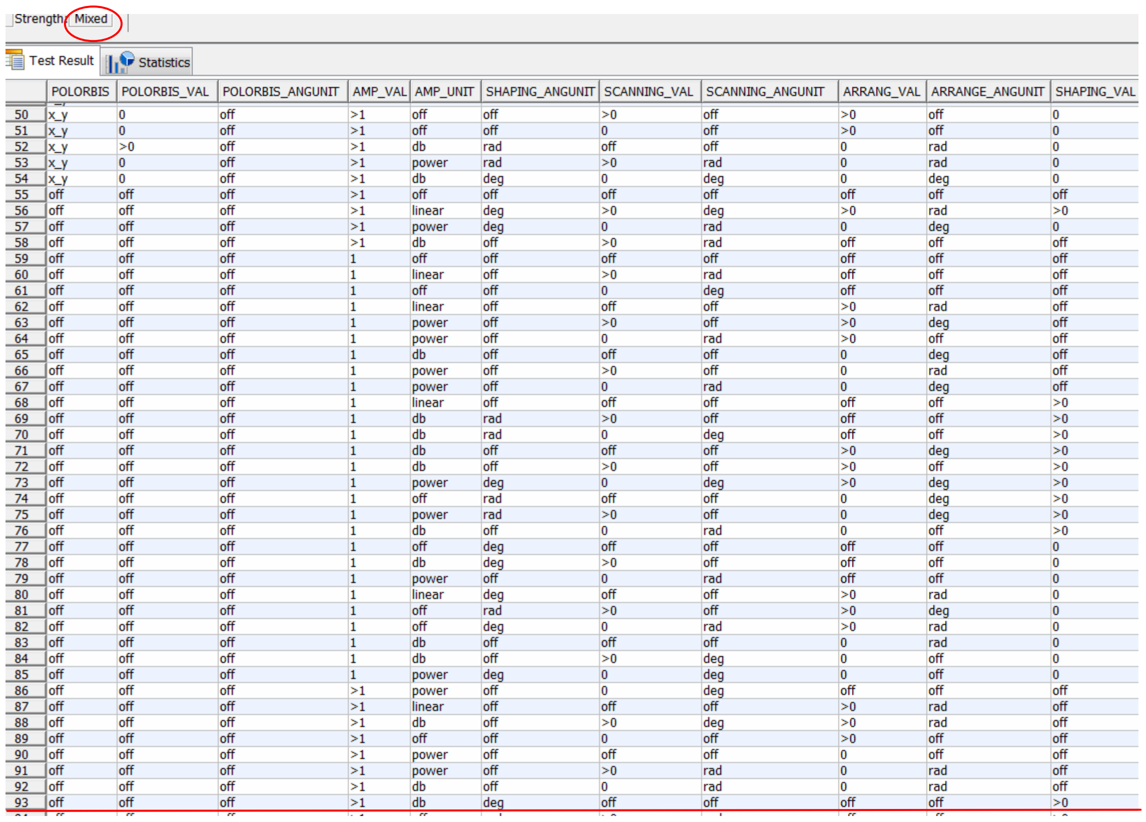
| | |
|---|---|
| Type = circular<br>angle_unit = NA<br>Circular_type = LH | Orientation = NA<br>Orientation _value = NA |

#### 4.5.3.1.4 Ports

Program can have one or more ports. For each individual port, the port definition is available. The test factors of an individual PORT are shown in Table 4.41.

Table 4.41 Test factors of individual port

| Test factors | Test values |
|---|---|
| Polorbis | [off, x_y, angle] |
| polorbis_val | [off, >0, 0] |
| polorbis_angunit | [off, rad, deg] |
| Amp_val | [1, >1] |
| Amp_unit | [off, linear, power, db] |
| shaping_val | [off, >0, 0] |
| shaping_angunit | [off, rad, deg] |
| scanning_val | [off, >0, 0] |
| scanning_angunit | [off, rad, deg] |
| arrang_val | [off, >0, 0] |
| arrange_angunit | [off, rad, deg] |



| | POLORBIS | POLORBIS_VAL | POLORBIS_ANGUNIT | AMP_VAL | AMP_UNIT | SHAPING_ANGUNIT | SCANNING_VAL | SCANNING_ANGUNIT | ARRANG_VAL | ARRANGE_ANGUNIT | SHAPING_VAL |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 50 | x_y | 0 | off | >1 | off | off | >0 | off | >0 | off | 0 |
| 51 | x_y | 0 | off | >1 | off | off | 0 | off | >0 | off | 0 |
| 52 | x_y | >0 | off | >1 | db | rad | off | off | 0 | rad | 0 |
| 53 | x_y | 0 | off | >1 | power | rad | >0 | rad | 0 | rad | 0 |
| 54 | x_y | 0 | off | >1 | db | deg | 0 | deg | 0 | deg | 0 |
| 55 | off | off | off | >1 | off | off | off | off | off | off | off |
| 56 | off | off | off | >1 | linear | deg | >0 | deg | >0 | rad | >0 |
| 57 | off | off | off | >1 | power | deg | 0 | rad | 0 | deg | 0 |
| 58 | off | off | off | >1 | db | off | >0 | rad | off | off | off |
| 59 | off | off | off | 1 | off | off | off | off | off | off | off |
| 60 | off | off | off | 1 | linear | off | >0 | rad | off | off | off |
| 61 | off | off | off | 1 | off | off | 0 | deg | off | off | off |
| 62 | off | off | off | 1 | linear | off | off | off | >0 | rad | off |
| 63 | off | off | off | 1 | power | off | >0 | off | >0 | deg | off |
| 64 | off | off | off | 1 | power | off | 0 | rad | >0 | off | off |
| 65 | off | off | off | 1 | db | off | off | off | 0 | deg | off |
| 66 | off | off | off | 1 | power | off | >0 | off | 0 | rad | off |
| 67 | off | off | off | 1 | power | off | 0 | rad | 0 | deg | off |
| 68 | off | off | off | 1 | linear | off | off | off | off | off | >0 |
| 69 | off | off | off | 1 | db | rad | >0 | off | off | off | >0 |
| 70 | off | off | off | 1 | db | rad | 0 | deg | off | off | >0 |
| 71 | off | off | off | 1 | db | off | off | off | >0 | deg | >0 |
| 72 | off | off | off | 1 | db | off | >0 | off | >0 | off | >0 |
| 73 | off | off | off | 1 | power | deg | 0 | deg | >0 | deg | >0 |
| 74 | off | off | off | 1 | off | rad | off | off | 0 | deg | >0 |
| 75 | off | off | off | 1 | power | rad | >0 | off | 0 | deg | >0 |
| 76 | off | off | off | 1 | db | off | 0 | rad | 0 | off | >0 |
| 77 | off | off | off | 1 | off | deg | off | off | off | off | 0 |
| 78 | off | off | off | 1 | db | deg | >0 | off | off | off | 0 |
| 79 | off | off | off | 1 | power | off | 0 | rad | off | off | 0 |
| 80 | off | off | off | 1 | linear | deg | off | off | >0 | rad | 0 |
| 81 | off | off | off | 1 | off | rad | >0 | off | >0 | deg | 0 |
| 82 | off | off | off | 1 | off | deg | 0 | rad | >0 | rad | 0 |
| 83 | off | off | off | 1 | db | off | off | off | 0 | rad | 0 |
| 84 | off | off | off | 1 | db | off | >0 | deg | 0 | off | 0 |
| 85 | off | off | off | 1 | power | deg | 0 | deg | 0 | off | 0 |
| 86 | off | off | off | >1 | power | off | 0 | deg | off | off | off |
| 87 | off | off | off | >1 | linear | off | off | off | >0 | rad | off |
| 88 | off | off | off | >1 | db | off | >0 | deg | >0 | rad | off |
| 89 | off | off | off | >1 | off | off | 0 | off | >0 | off | off |
| 90 | off | off | off | >1 | power | off | off | off | 0 | off | off |
| 91 | off | off | off | >1 | power | off | >0 | rad | 0 | rad | off |
| 92 | off | off | off | >1 | db | off | 0 | rad | 0 | rad | off |
| 93 | off | off | off | >1 | db | deg | off | off | off | off | >0 |

Figure 4.20 A part of test cases of port
Total 162 test cases

The value of the 'ampval' parameter is the value of the amplitude of the port which is a required parameter and other parameters are all optional.

The following will show a concrete value for the abstract test case # 93 which defines one port:

```
PORT 1
    AMPLITUDE 10 db
    PHASE_SHAPING 360 deg
```

| | |
|---|---|
| Polorbis = off<br>polorbis_angval =off<br>polorbis_angunit = off<br>Amp_val = >1<br>Amp_unit = db<br>shaping_val = >0 | shaping_angunit = deg<br>scanning_val = off<br>scanning_angunit = off<br>arrange_val = off<br>arrange_angunit = off |

In addition, program can have multiple ports. We also need to identify a parameter indicate the number of ports that we should have in the ADL file. Each port also has a unique identifier. We took care of the id and number of ports while we are generating our concrete test case. We will discuss this in the following paragraphs.

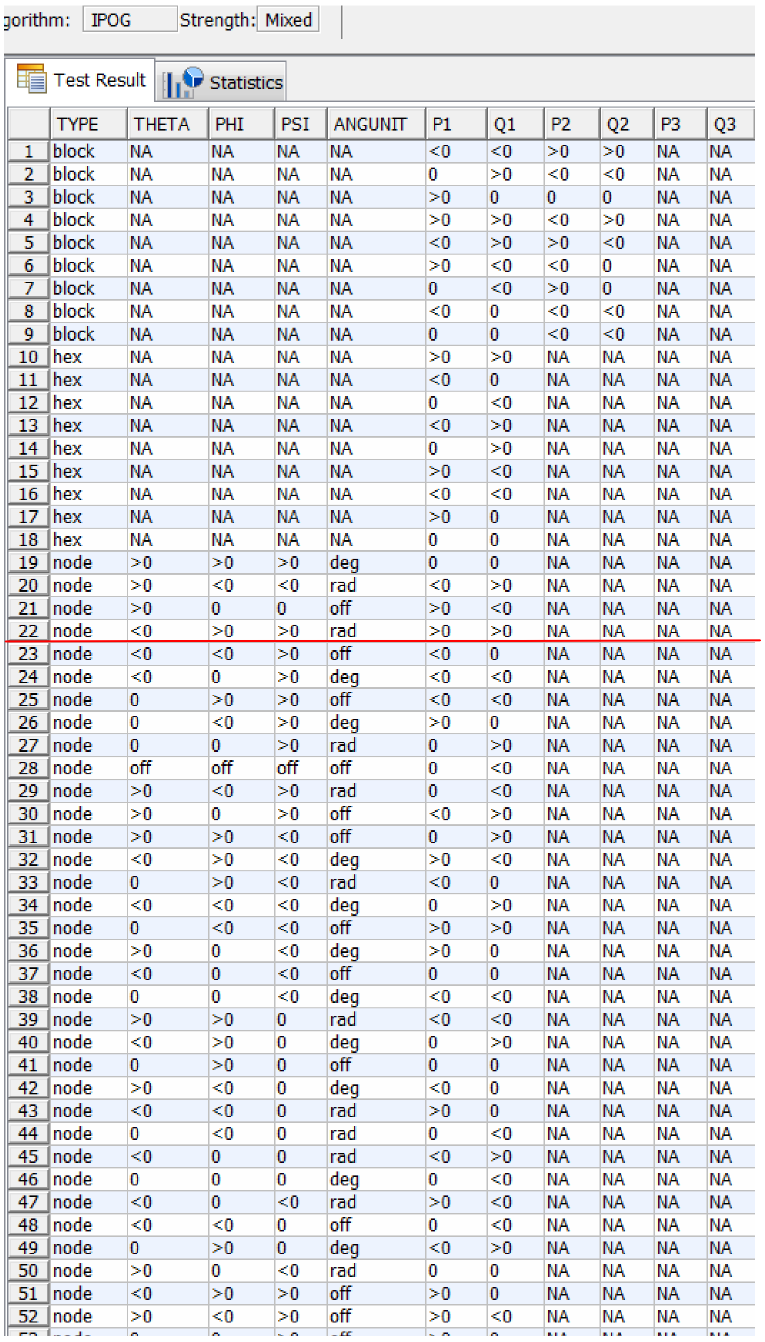
##### 4.5.3.1.5 Add_remove

In *space* program, we can add and remove four types of object such as node, block, polygon and hexagon. If we didn't add any object we cannot call the remove; therefore, if we are going to call remove we need to call it after add. Also we can have multiple add and remove. In addition add is a required parameter but remove is optional. But since they call exactly the same object definitions, we modeled them together.

The parameters "number_of_add" and "number_of_remove" indicate the number of times the 'add' or 'remove' will be called and they will be assigned to the GROUP model.

Table 4.42 Test factors of add_remove

| Test factors | Test values |
|---|---|
| Type | [node, block, poly, hex] |
| THETA | [NA, off,>0, <0, 0] |
| PHI | [NA, off,>0, <0, 0] |
| PSI | [NA, off,>0, <0, 0] |
| Angunit | [NA, off, rad, deg] |
| P1 | [ >0, <0, 0] |
| Q1 | [ >0, <0, 0] |
| P2 | [NA, >0, <0, 0] |
| Q2 | [NA, >0, <0, 0] |
| P3 | [NA, off,>0, <0, 0] |
| Q3 | [NA, off,>0, <0, 0] |



gorithm: IPOG Strength: Mixed

Test Result | Statistics

| | TYPE | THETA | PHI | PSI | ANGUNIT | P1 | Q1 | P2 | Q2 | P3 | Q3 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | block | NA | NA | NA | NA | <0 | <0 | >0 | >0 | NA | NA |
| 2 | block | NA | NA | NA | NA | 0 | >0 | <0 | <0 | NA | NA |
| 3 | block | NA | NA | NA | NA | >0 | 0 | 0 | 0 | NA | NA |
| 4 | block | NA | NA | NA | NA | >0 | >0 | <0 | >0 | NA | NA |
| 5 | block | NA | NA | NA | NA | <0 | >0 | >0 | <0 | NA | NA |
| 6 | block | NA | NA | NA | NA | >0 | <0 | <0 | 0 | NA | NA |
| 7 | block | NA | NA | NA | NA | 0 | <0 | >0 | 0 | NA | NA |
| 8 | block | NA | NA | NA | NA | <0 | 0 | <0 | <0 | NA | NA |
| 9 | block | NA | NA | NA | NA | 0 | 0 | <0 | <0 | NA | NA |
| 10 | hex | NA | NA | NA | NA | >0 | >0 | NA | NA | NA | NA |
| 11 | hex | NA | NA | NA | NA | <0 | 0 | NA | NA | NA | NA |
| 12 | hex | NA | NA | NA | NA | 0 | <0 | NA | NA | NA | NA |
| 13 | hex | NA | NA | NA | NA | <0 | >0 | NA | NA | NA | NA |
| 14 | hex | NA | NA | NA | NA | 0 | >0 | NA | NA | NA | NA |
| 15 | hex | NA | NA | NA | NA | >0 | <0 | NA | NA | NA | NA |
| 16 | hex | NA | NA | NA | NA | <0 | <0 | NA | NA | NA | NA |
| 17 | hex | NA | NA | NA | NA | >0 | 0 | NA | NA | NA | NA |
| 18 | hex | NA | NA | NA | NA | 0 | 0 | NA | NA | NA | NA |
| 19 | node | >0 | >0 | >0 | deg | 0 | 0 | NA | NA | NA | NA |
| 20 | node | >0 | <0 | <0 | rad | <0 | >0 | NA | NA | NA | NA |
| 21 | node | >0 | 0 | 0 | off | >0 | <0 | NA | NA | NA | NA |
| 22 | node | <0 | >0 | >0 | rad | >0 | >0 | NA | NA | NA | NA |
| 23 | node | <0 | <0 | >0 | off | <0 | 0 | NA | NA | NA | NA |
| 24 | node | <0 | 0 | >0 | deg | <0 | <0 | NA | NA | NA | NA |
| 25 | node | 0 | >0 | >0 | off | <0 | <0 | NA | NA | NA | NA |
| 26 | node | 0 | <0 | >0 | deg | >0 | 0 | NA | NA | NA | NA |
| 27 | node | 0 | 0 | >0 | rad | 0 | >0 | NA | NA | NA | NA |
| 28 | node | off | off | off | off | 0 | <0 | NA | NA | NA | NA |
| 29 | node | >0 | <0 | >0 | rad | 0 | <0 | NA | NA | NA | NA |
| 30 | node | >0 | 0 | >0 | off | <0 | >0 | NA | NA | NA | NA |
| 31 | node | >0 | >0 | <0 | off | 0 | >0 | NA | NA | NA | NA |
| 32 | node | <0 | >0 | <0 | deg | >0 | <0 | NA | NA | NA | NA |
| 33 | node | 0 | >0 | <0 | rad | <0 | 0 | NA | NA | NA | NA |
| 34 | node | <0 | <0 | <0 | deg | 0 | >0 | NA | NA | NA | NA |
| 35 | node | 0 | <0 | <0 | off | >0 | >0 | NA | NA | NA | NA |
| 36 | node | >0 | 0 | <0 | deg | >0 | 0 | NA | NA | NA | NA |
| 37 | node | <0 | 0 | <0 | off | 0 | 0 | NA | NA | NA | NA |
| 38 | node | 0 | 0 | <0 | deg | <0 | <0 | NA | NA | NA | NA |
| 39 | node | >0 | >0 | 0 | rad | <0 | <0 | NA | NA | NA | NA |
| 40 | node | <0 | >0 | 0 | deg | 0 | >0 | NA | NA | NA | NA |
| 41 | node | 0 | >0 | 0 | off | 0 | 0 | NA | NA | NA | NA |
| 42 | node | >0 | <0 | 0 | deg | <0 | 0 | NA | NA | NA | NA |
| 43 | node | <0 | <0 | 0 | rad | >0 | 0 | NA | NA | NA | NA |
| 44 | node | 0 | <0 | 0 | rad | 0 | <0 | NA | NA | NA | NA |
| 45 | node | <0 | 0 | 0 | rad | <0 | >0 | NA | NA | NA | NA |
| 46 | node | 0 | 0 | 0 | deg | 0 | <0 | NA | NA | NA | NA |
| 47 | node | <0 | 0 | <0 | rad | >0 | <0 | NA | NA | NA | NA |
| 48 | node | <0 | <0 | 0 | off | 0 | <0 | NA | NA | NA | NA |
| 49 | node | 0 | >0 | 0 | deg | <0 | >0 | NA | NA | NA | NA |
| 50 | node | >0 | 0 | <0 | rad | 0 | 0 | NA | NA | NA | NA |
| 51 | node | <0 | >0 | >0 | off | >0 | 0 | NA | NA | NA | NA |
| 52 | node | >0 | <0 | >0 | off | >0 | <0 | NA | NA | NA | NA |

Figure 4.21 Part of test cases of add_remove
Total 91 test cases

The following will show a concrete value for the abstract test case #22:

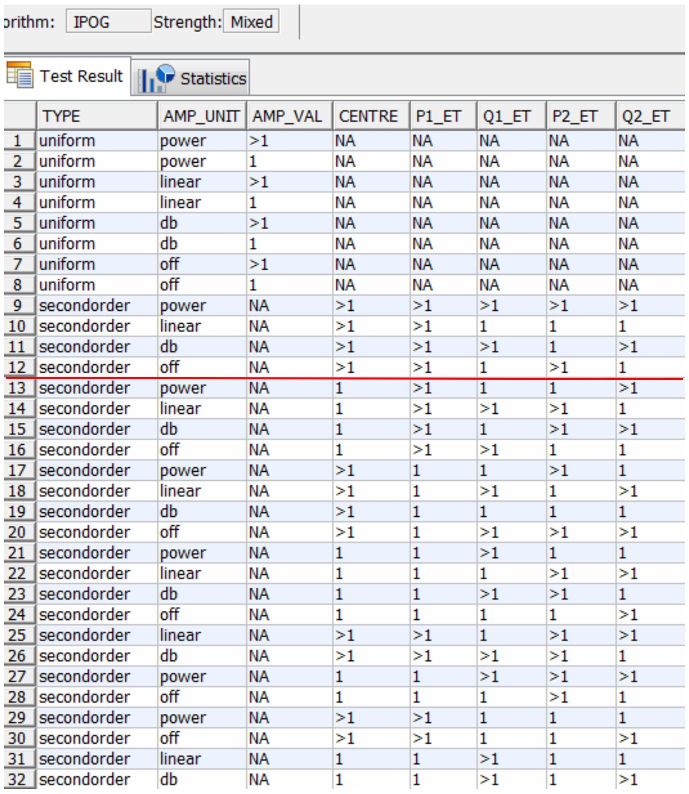
ADD NODE 2 2 ORIENTATION -2 2 1 rad

| | |
|---|---|
| type = node<br>THETA = <0<br>PHI = >0<br>PSI = >0<br>angunit = rad<br>P1 = >0 | Q1 = >0<br>P2 = NA<br>Q2 = NA<br>P3 = NA<br>Q3 = NA |

#### 4.5.3.1.6 Amplitude

Table 4.43 Test factors of amplitude

| Test factors | Test values |
|---|---|
| Type | [uniform, secondorder] |
| Amp_val | [NA, >1, 1] |
| Amp_unit | [off, power, linear, dB] |
| CENTRE | [NA, >1, 1] |
| P1_ET | [NA, >1, 1] |
| Q1_ET | [NA, >1, 1] |
| P2_ET | [NA, >1, 1] |
| Q2_ET | [NA, >1, 1] |



| | TYPE | AMP_UNIT | AMP_VAL | CENTRE | P1_ET | Q1_ET | P2_ET | Q2_ET |
|---|---|---|---|---|---|---|---|---|
| 1 | uniform | power | >1 | NA | NA | NA | NA | NA |
| 2 | uniform | power | 1 | NA | NA | NA | NA | NA |
| 3 | uniform | linear | >1 | NA | NA | NA | NA | NA |
| 4 | uniform | linear | 1 | NA | NA | NA | NA | NA |
| 5 | uniform | db | >1 | NA | NA | NA | NA | NA |
| 6 | uniform | db | 1 | NA | NA | NA | NA | NA |
| 7 | uniform | off | >1 | NA | NA | NA | NA | NA |
| 8 | uniform | off | 1 | NA | NA | NA | NA | NA |
| 9 | secondorder | power | NA | >1 | >1 | >1 | >1 | >1 |
| 10 | secondorder | linear | NA | >1 | >1 | 1 | 1 | 1 |
| 11 | secondorder | db | NA | >1 | >1 | >1 | 1 | >1 |
| 12 | secondorder | off | NA | >1 | >1 | 1 | >1 | 1 |
| 13 | secondorder | power | NA | 1 | >1 | 1 | 1 | >1 |
| 14 | secondorder | linear | NA | 1 | >1 | >1 | >1 | 1 |
| 15 | secondorder | db | NA | 1 | >1 | 1 | >1 | >1 |
| 16 | secondorder | off | NA | 1 | >1 | >1 | 1 | 1 |
| 17 | secondorder | power | NA | >1 | 1 | 1 | >1 | 1 |
| 18 | secondorder | linear | NA | >1 | 1 | >1 | 1 | >1 |
| 19 | secondorder | db | NA | >1 | 1 | 1 | 1 | 1 |
| 20 | secondorder | off | NA | >1 | 1 | >1 | >1 | >1 |
| 21 | secondorder | power | NA | 1 | 1 | >1 | 1 | 1 |
| 22 | secondorder | linear | NA | 1 | 1 | 1 | >1 | >1 |
| 23 | secondorder | db | NA | 1 | 1 | >1 | >1 | 1 |
| 24 | secondorder | off | NA | 1 | 1 | 1 | 1 | >1 |
| 25 | secondorder | linear | NA | >1 | >1 | 1 | >1 | >1 |
| 26 | secondorder | db | NA | >1 | >1 | >1 | >1 | 1 |
| 27 | secondorder | power | NA | 1 | 1 | >1 | >1 | >1 |
| 28 | secondorder | off | NA | 1 | 1 | 1 | >1 | 1 |
| 29 | secondorder | power | NA | >1 | >1 | 1 | 1 | 1 |
| 30 | secondorder | off | NA | >1 | >1 | 1 | 1 | >1 |
| 31 | secondorder | linear | NA | 1 | 1 | >1 | 1 | 1 |
| 32 | secondorder | db | NA | 1 | 1 | >1 | 1 | >1 |

Figure 4.22 Test cases of amplitude

The following will show a concrete value for the abstract test case #12:

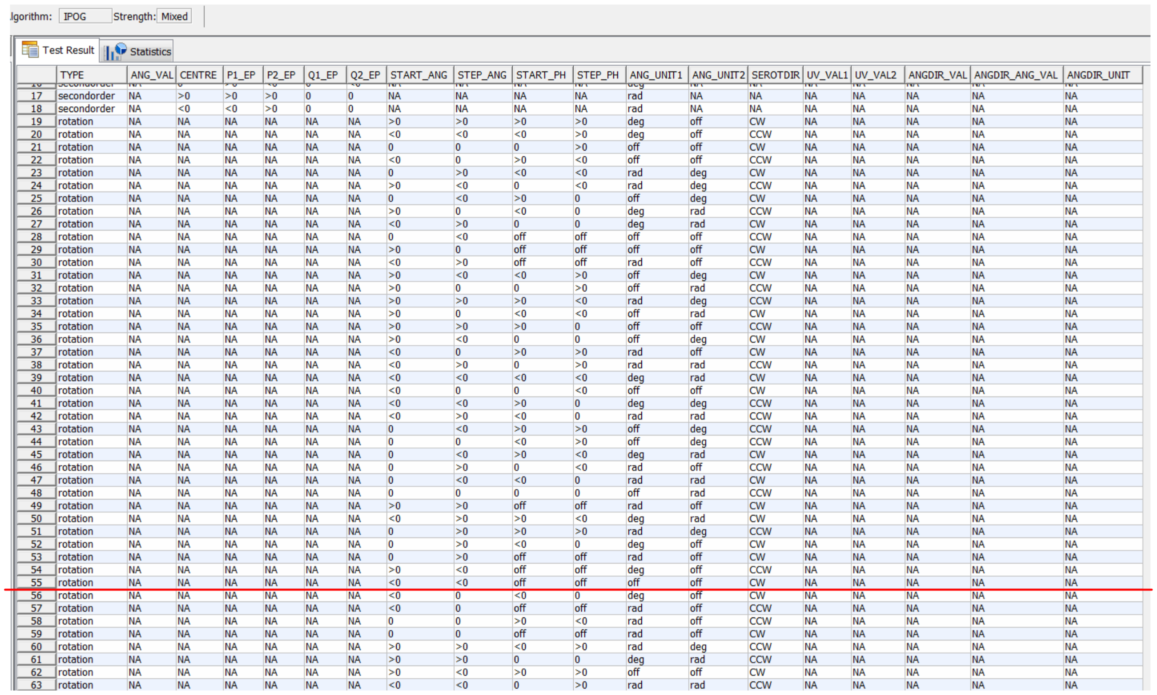
AMPLITUDE SECOND ORDER 20 21 1 22 23

| | |
|---|---|
| Type = secondorder<br>Amp_val = NA<br>Amp_unit = off<br>CENTRE = >1 | P1_ET = >1<br>Q1_ET = >1<br>P2_ET = >1<br>Q2_ET = >1 |

#### 4.5.3.1.7 Phase

Table 4.44 Test factors of phase

| Test factors | Test values |
|---|---|
| Type | [uniform, secondorder, rotation, pointing] |
| Ang_val | [NA, >0, <0, 0] |
| CENTRE | [NA, >0, <0, 0] |
| P1_EP | [NA, >0, <0, 0] |
| P2_EP | [NA ,>0, <0, 0] |
| Q1_EP | [NA ,>0, <0, 0] |
| Q2_EP | [NA, >0, <0, 0] |
| Start_ang | [NA, >0, <0, 0] |
| Step_ang | [NA, >0, <0, 0] |
| Start_ph | [NA, off, >0, <0, 0] |
| Step_ph | [NA, off, >0, <0, 0] |
| Ang_unit1 | [NA,off, rad, deg] |
| Ang_unit2 | [NA, off, rad, deg] |
| Serotdir | [NA, CW,CCW] |
| Uv_val1 | [NA,-1, -0.5, 0, 0.5, 1] |
| Uv_val2 | [NA,-1, -0.5, 0, 0.5, 1] |
| Angdir_val | [NA ,>0, <0, 0] |
| Angdir_ang_val | [NA ,>0, <0, 0] |
| Angdir_unit | [NA, off, rad, deg] |


lgorithm: IPOG Strength: Mixed

Test Result | Statistics

| | TYPE | ANG_VAL | CENTRE | P1_EP | P2_EP | Q1_EP | Q2_EP | START_ANG | STEP_ANG | START_PH | STEP_PH | ANG_UNIT1 | ANG_UNIT2 | SEROTDIR | UV_VAL1 | UV_VAL2 | ANGDIR_VAL | ANGDIR_ANG_VAL | ANGDIR_UNIT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 17 | secondorder | NA | >0 | >0 | >0 | 0 | 0 | NA | NA | NA | NA | rad | NA | NA | NA | NA | NA | NA | NA |
| 18 | secondorder | NA | <0 | <0 | >0 | 0 | 0 | NA | NA | NA | NA | rad | NA | NA | NA | NA | NA | NA | NA |
| 19 | rotation | NA | NA | NA | NA | NA | NA | >0 | >0 | >0 | >0 | deg | off | CW | NA | NA | NA | NA | NA |
| 20 | rotation | NA | NA | NA | NA | NA | NA | <0 | <0 | <0 | >0 | deg | off | CCW | NA | NA | NA | NA | NA |
| 21 | rotation | NA | NA | NA | NA | NA | NA | 0 | 0 | 0 | >0 | off | off | CW | NA | NA | NA | NA | NA |
| 22 | rotation | NA | NA | NA | NA | NA | NA | <0 | 0 | >0 | <0 | off | off | CCW | NA | NA | NA | NA | NA |
| 23 | rotation | NA | NA | NA | NA | NA | NA | 0 | >0 | <0 | <0 | rad | deg | CW | NA | NA | NA | NA | NA |
| 24 | rotation | NA | NA | NA | NA | NA | NA | >0 | <0 | 0 | <0 | rad | deg | CCW | NA | NA | NA | NA | NA |
| 25 | rotation | NA | NA | NA | NA | NA | NA | 0 | <0 | >0 | 0 | off | deg | CW | NA | NA | NA | NA | NA |
| 26 | rotation | NA | NA | NA | NA | NA | NA | >0 | 0 | <0 | 0 | deg | rad | CCW | NA | NA | NA | NA | NA |
| 27 | rotation | NA | NA | NA | NA | NA | NA | <0 | >0 | 0 | 0 | deg | rad | CW | NA | NA | NA | NA | NA |
| 28 | rotation | NA | NA | NA | NA | NA | NA | 0 | <0 | off | off | off | off | CCW | NA | NA | NA | NA | NA |
| 29 | rotation | NA | NA | NA | NA | NA | NA | >0 | 0 | off | off | off | off | CW | NA | NA | NA | NA | NA |
| 30 | rotation | NA | NA | NA | NA | NA | NA | <0 | >0 | off | off | rad | off | CCW | NA | NA | NA | NA | NA |
| 31 | rotation | NA | NA | NA | NA | NA | NA | >0 | <0 | <0 | >0 | off | deg | CW | NA | NA | NA | NA | NA |
| 32 | rotation | NA | NA | NA | NA | NA | NA | >0 | 0 | 0 | >0 | off | rad | CCW | NA | NA | NA | NA | NA |
| 33 | rotation | NA | NA | NA | NA | NA | NA | >0 | >0 | >0 | <0 | rad | deg | CCW | NA | NA | NA | NA | NA |
| 34 | rotation | NA | NA | NA | NA | NA | NA | >0 | 0 | <0 | <0 | off | rad | CW | NA | NA | NA | NA | NA |
| 35 | rotation | NA | NA | NA | NA | NA | NA | >0 | >0 | >0 | 0 | off | off | CCW | NA | NA | NA | NA | NA |
| 36 | rotation | NA | NA | NA | NA | NA | NA | >0 | <0 | 0 | 0 | off | deg | CW | NA | NA | NA | NA | NA |
| 37 | rotation | NA | NA | NA | NA | NA | NA | <0 | 0 | >0 | >0 | rad | off | CW | NA | NA | NA | NA | NA |
| 38 | rotation | NA | NA | NA | NA | NA | NA | <0 | >0 | 0 | >0 | rad | rad | CCW | NA | NA | NA | NA | NA |
| 39 | rotation | NA | NA | NA | NA | NA | NA | <0 | <0 | <0 | <0 | deg | rad | CW | NA | NA | NA | NA | NA |
| 40 | rotation | NA | NA | NA | NA | NA | NA | <0 | 0 | 0 | <0 | off | off | CW | NA | NA | NA | NA | NA |
| 41 | rotation | NA | NA | NA | NA | NA | NA | <0 | <0 | >0 | 0 | deg | deg | CCW | NA | NA | NA | NA | NA |
| 42 | rotation | NA | NA | NA | NA | NA | NA | <0 | >0 | <0 | 0 | rad | rad | CCW | NA | NA | NA | NA | NA |
| 43 | rotation | NA | NA | NA | NA | NA | NA | 0 | <0 | >0 | >0 | off | deg | CCW | NA | NA | NA | NA | NA |
| 44 | rotation | NA | NA | NA | NA | NA | NA | 0 | 0 | <0 | >0 | off | deg | CCW | NA | NA | NA | NA | NA |
| 45 | rotation | NA | NA | NA | NA | NA | NA | 0 | <0 | >0 | <0 | deg | rad | CW | NA | NA | NA | NA | NA |
| 46 | rotation | NA | NA | NA | NA | NA | NA | 0 | >0 | 0 | <0 | rad | off | CCW | NA | NA | NA | NA | NA |
| 47 | rotation | NA | NA | NA | NA | NA | NA | 0 | <0 | <0 | 0 | rad | rad | CW | NA | NA | NA | NA | NA |
| 48 | rotation | NA | NA | NA | NA | NA | NA | 0 | 0 | 0 | 0 | off | rad | CCW | NA | NA | NA | NA | NA |
| 49 | rotation | NA | NA | NA | NA | NA | NA | >0 | >0 | off | off | rad | off | CW | NA | NA | NA | NA | NA |
| 50 | rotation | NA | NA | NA | NA | NA | NA | <0 | >0 | >0 | <0 | deg | rad | CW | NA | NA | NA | NA | NA |
| 51 | rotation | NA | NA | NA | NA | NA | NA | 0 | >0 | >0 | >0 | rad | deg | CCW | NA | NA | NA | NA | NA |
| 52 | rotation | NA | NA | NA | NA | NA | NA | 0 | >0 | <0 | 0 | deg | off | CW | NA | NA | NA | NA | NA |
| 53 | rotation | NA | NA | NA | NA | NA | NA | 0 | >0 | off | off | rad | off | CW | NA | NA | NA | NA | NA |
| 54 | rotation | NA | NA | NA | NA | NA | NA | >0 | <0 | off | off | deg | off | CCW | NA | NA | NA | NA | NA |
| 55 | rotation | NA | NA | NA | NA | NA | NA | <0 | <0 | off | off | off | off | CW | NA | NA | NA | NA | NA |
| 56 | rotation | NA | NA | NA | NA | NA | NA | <0 | 0 | <0 | 0 | deg | off | CW | NA | NA | NA | NA | NA |
| 57 | rotation | NA | NA | NA | NA | NA | NA | <0 | 0 | off | off | rad | off | CCW | NA | NA | NA | NA | NA |
| 58 | rotation | NA | NA | NA | NA | NA | NA | 0 | 0 | >0 | <0 | rad | off | CCW | NA | NA | NA | NA | NA |
| 59 | rotation | NA | NA | NA | NA | NA | NA | 0 | 0 | off | off | rad | off | CW | NA | NA | NA | NA | NA |
| 60 | rotation | NA | NA | NA | NA | NA | NA | >0 | >0 | <0 | >0 | rad | deg | CCW | NA | NA | NA | NA | NA |
| 61 | rotation | NA | NA | NA | NA | NA | NA | >0 | >0 | 0 | 0 | deg | rad | CCW | NA | NA | NA | NA | NA |
| 62 | rotation | NA | NA | NA | NA | NA | NA | >0 | <0 | >0 | >0 | off | off | CW | NA | NA | NA | NA | NA |
| 63 | rotation | NA | NA | NA | NA | NA | NA | <0 | <0 | 0 | >0 | rad | rad | CCW | NA | NA | NA | NA | NA |

Figure 4.23 Part of test cases of phase
Total 100 test cases

The following will show a concrete value for an abstract test case #55:

PHASE ROTATION SEQUENTIAL CW ANGLE 20 20

| | |
|---|---|
| Type = rotation<br>Angval = NA<br>CENTRE = NA<br>P1_EP = NA<br>P2_EP = NA<br>Q1_EP = NA<br>Q2_EP = NA<br>Start_ang = >0<br>Step_ang = >0<br>Start_ph = off | Step_ph = off<br>Ang_unit1 = off<br>Ang_unit2 = off<br>Serotdir = CW<br>Uv_val1 = NA<br>Uv_val2 = NA<br>Angdir_val = NA<br>Angdir_angval = NA<br>Angdir_unit = NA |

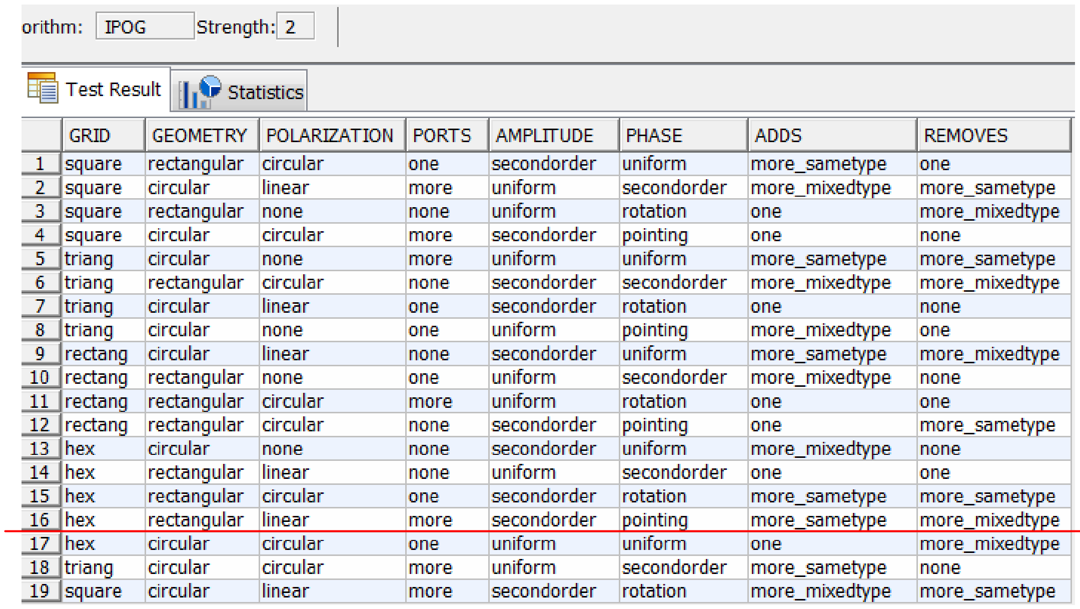
### 4.5.3.1.8 Group

As it is shown in Figure 4.16, a group consists of three components. Now that we modeled all the components of the GROUP, it is time to model the GROUP itself.

Some of the components in our graph model act like a switch for the other components. They don't have any other functionality or attributes. For this kind of components, we don't have to model them and instead we can add parameters to the parent node. We ignore the element, grid_element, and excitation components from our model and instead we add all the needed information to the parent node which is the Group as it is shown in Table 4.45.

Table 4.45 Test factors of group

| Test factors | Test values |
|---|---|
| Grid | [none, square, triang, rectang, hex] |
| Polarization | [none, circular, linear] |
| geometry | [none, circular, rectangular] |
| Ports | [ none, one, more] |
| number_of_add | [one, more_sametype, more_mixedtype] |
| number_of_remove | [none, one, more] |
| Amplitude | [uniform, secondorder] |
| phase | [uniform, secondorder, rotation, pointing] |


orithm: IPOG Strength: 2

Test Result | Statistics

|  | GRID | GEOMETRY | POLARIZATION | PORTS | AMPLITUDE | PHASE | ADDS | REMOVES |
|---|---|---|---|---|---|---|---|---|
| 1 | square | rectangular | circular | one | secondorder | uniform | more_sametype | one |
| 2 | square | circular | linear | more | uniform | secondorder | more_mixedtype | more_sametype |
| 3 | square | rectangular | none | none | uniform | rotation | one | more_mixedtype |
| 4 | square | circular | circular | more | secondorder | pointing | one | none |
| 5 | triang | circular | none | more | uniform | uniform | more_sametype | more_sametype |
| 6 | triang | rectangular | circular | none | secondorder | secondorder | more_mixedtype | more_mixedtype |
| 7 | triang | circular | linear | one | secondorder | rotation | one | none |
| 8 | triang | circular | none | one | uniform | pointing | more_mixedtype | one |
| 9 | rectang | circular | linear | none | secondorder | uniform | more_sametype | more_mixedtype |
| 10 | rectang | rectangular | none | one | uniform | secondorder | more_mixedtype | none |
| 11 | rectang | rectangular | circular | more | uniform | rotation | one | one |
| 12 | rectang | rectangular | circular | none | secondorder | pointing | one | more_sametype |
| 13 | hex | circular | none | none | secondorder | uniform | more_mixedtype | none |
| 14 | hex | rectangular | linear | none | uniform | secondorder | one | one |
| 15 | hex | rectangular | circular | one | secondorder | rotation | more_sametype | more_sametype |
| 16 | hex | rectangular | linear | more | secondorder | pointing | more_sametype | more_mixedtype |
| 17 | hex | circular | circular | one | uniform | uniform | one | more_mixedtype |
| 18 | triang | circular | circular | more | uniform | secondorder | more_sametype | none |
| 19 | square | circular | linear | more | secondorder | rotation | more_mixedtype | more_sametype |

Figure 4.24 2-way test cases of group

Figure 4.24 shows the 2-way test cases of Group. We want to cover all the different types of the components in a group. The concrete test case generator will pick one of the test cases that are already generated for each of these components. The Table 4.46 will show an example of concrete test case for the abstract test case #16 which defines a group contains a hexagonal grid, one element that contains a rectangular geometry, a linear polarization and four ports. In this concrete value we covered test case #7 of gird, test case #5 of geometry, test case

#4 of polarization and test cases #55, #59 ,#90 and #87 of ports. The reason we have four ports is that in the abstract test case the value of ports is “more”. When we generate concrete values, we convert this to a value between 2 to 9.

Also the value of adds in the abstract test case is “more_sametype” and the value of removes is “more_mixedtype”. This means that we need to have two or more add of similar types and two or more remove of different types in the concrete test case. As we mentioned before, there are four types of add and remove: block, node, polygon and hexagon. Therefore, the concrete value for “more_sametype add” is four “add” all block types and the concrete value for “more_mixedtype remove” is four “remove” one for each type. The numbers of covered test cases are written next to each line.

Table 4.46 A sample concrete test case (an ADL file)

```
GROUP testgroup
        GRID HEXAGONAL  STEP 40 m //test case #7
        ELEMENT
                GEOMETRY RECTANGULAR  20 40 //test case #5
                POLARIZATION LINEAR ORIENTATION 90 deg //test case #4
        PORTS 4
                PORT 1 // test case # 55
                    AMPLITUDE 10
                PORT 2 //test case # 90
                    AMPLITUDE 10 POWER
                    PHASE_POL_ARRANG 0
                PORT 3 // test case # 59
                    AMPLITUDE 1
                PORT 4 // test case # 87
                    AMPLITUDE 10 LINEAR
                    PHASE_POL_ARRANG 20 rad
        ADD             BLOCK -2 -2 1 1 // test case # 1
        ADD             BLOCK 1 -2 -2  0 // test case # 6
        ADD             BLOCK 1 1 -2 1 // test case # 4
        ADD             BLOCK -2 0 -2 -2 // test case # 8
        REMOVE          BLOCK 1 1 2 2 // test case # 1
        REMOVE          POLYGON 1 1 2 -2 -1 -1 // test case # 81
        REMOVE          HEXAGON 1 1  // test case # 10
        REMOVE          NODE 1 2 // test case # 66
        GROUP_EXCITATION
                AMPLITUDE       SECOND ORDER 27 32 26 18 17 power// test case # 9
                PHASE           POINTING U 0.5 V 0.5   // test case # 85
END
```

Finally the abstract value of amplitude and phase are ‘secondorder’ and ‘pointing’. Therefore, the concrete test case contains one second order amplitude and one pointing phase. Again the numbers of covered test cases are written next to each line.

Number of 3-way test cases for the GROUP are 74. To cover all the different possible cases, 4 to 6 concrete test cases for each abstract test case are generated. Therefore, total 120 concrete test cases are generated for 2-way testing and 315 for 3-way testing.

Table 4.47 Summary of space model

| Application Space | # of parameter values | # of parameter | # of constraint parameters | # of constraint | # of relation parameters | # of relation |
|---|---|---|---|---|---|---|
| GRID | $3^{13}4^{2}$ | 15 | $2^{3}3^{4}4^{2}5^{2}6^{3}7^{1}$ | 15 | $3^{1}$ | 1 |
| GEOMETRY | $2^{2}3^{3}4^{1}$ | 6 | $5^{2}$ | 2 | $2^{1}3^{1}$ | 2 |
| POLARIZATION | $2^{1}4^{4}$ | 5 | $3^{4}5^{1}$ | 5 | $4^{1}$ | 1 |
| PORTS | $2^{1}3^{9}4^{1}$ | 11 | $2^{6}3^{1}$ | 7 | $2^{4}4^{1}$ | 5 |
| ADD_REMOVE | $4^{6}5^{5}$ | 11 | $2^{1}4^{1}9^{4}$ | 6 | $2^{3}4^{1}$ | 4 |
| AMPLITUDE | $2^{1}3^{6}4^{1}$ | 8 | $7^{2}$ | 2 | $4^{2}$ | 2 |
| PHASE | $3^{12}4^{5}7^{2}$ | 19 | $2^{6}3^{5}14^{2}18^{2}$ | 15 | $2^{1}3^{2}4^{1}6^{1}$ | 5 |
| TOTAL | $2^{5}3^{43}4^{20}5^{5}7^{2}$ | 75 | $2^{16}3^{14}4^{3}5^{5}6^{3}7^{3}9^{4}14^{2}18^{2}$ | 52 | $2^{14}3^{4}4^{2}6^{1}$ | 20 |

4.5.3.2 Results and discussion

The designed model for Space has 7 IPMs which are shown in Table 4.47, yielding 315 generated test cases. Code coverage data are shown in Figure 4.25. We used *Gcov* to collect code coverage. Space statistics are shown in Table 4.48. e.g. number of lines of code in Space are 9126.

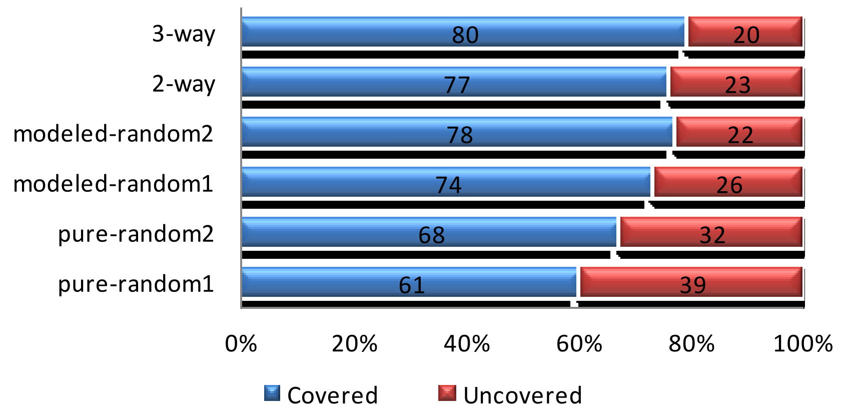


Figure 4.25 Space code coverage results

Gcov gives us the code coverage for all of the test cases. While we execute our tests, it highlights the parts of the source code that are executed. It is shown in Figure 4.25 that our tests cover more than 80% of lines.

Table 4.48 Space statistics

| | |
|---|---|
| LOC (line of code) | 9,126 |
| Number of Branches | 1,190 |
| Number of Functions | 136 |
| Number of faulty versions | 32 |
| Type of Faults | Real |

The total number of detected faults is 30. Table 4.49 shows the fault detection table for *Space*.

3-way testing was able to detect 93.7% of the faults. The faulty versions 12 and 18 (v12 and v18) were only killed by 3-way testing. None of our tests was able to detect v27. Version v33 was only detected by modeled-random testing (with the same number of tests as 2-way testing).

Table 4.49 Space fault detection table

| Version s | pure random1 | pure random2 | modeled random1 | modeled random2 | 2way | 3way |
|---|---|---|---|---|---|---|
| v4 | 1 | 1 | 1 | 1 | 1 | 1 |
| v5 | 1 | 1 | 1 | 1 | 1 | 1 |
| v6 | 1 | 1 | 1 | 1 | 1 | 1 |
| v7 | 0 | 0 | 0 | 0 | 1 | 1 |
| v8 | 0 | 0 | 0 | 0 | 1 | 1 |
| v9 | 0 | 0 | 1 | 1 | 1 | 1 |
| v10 | 0 | 0 | 1 | 1 | 1 | 1 |
| v11 | 0 | 1 | 1 | 1 | 1 | 1 |
| v12 | 0 | 0 | 0 | 0 | 0 | 1 |
| v13 | 0 | 1 | 1 | 1 | 1 | 1 |
| v14 | 1 | 1 | 1 | 1 | 1 | 1 |
| v15 | 1 | 1 | 1 | 1 | 1 | 1 |
| v16 | 0 | 0 | 1 | 1 | 1 | 1 |
| v17 | 1 | 1 | 1 | 1 | 1 | 1 |
| v18 | 0 | 0 | 0 | 0 | 0 | 1 |
| v19 | 0 | 0 | 1 | 1 | 1 | 1 |
| v20 | 0 | 0 | 1 | 1 | 1 | 1 |
| v21 | 0 | 0 | 1 | 1 | 1 | 1 |
| v22 | 0 | 0 | 0 | 1 | 1 | 1 |
| v23 | 0 | 0 | 1 | 1 | 1 | 1 |
| v24 | 1 | 1 | 1 | 1 | 1 | 1 |
| v25 | 1 | 1 | 1 | 1 | 1 | 1 |
| v26 | 1 | 1 | 1 | 1 | 1 | 1 |
| v27 | 0 | 0 | 0 | 0 | 0 | 0 |
| v28 | 1 | 1 | 1 | 1 | 1 | 1 |
| v29 | 0 | 0 | 1 | 1 | 1 | 1 |
| v30 | 1 | 1 | 1 | 1 | 1 | 1 |
| v31 | 1 | 1 | 1 | 1 | 1 | 1 |
| v33 | 0 | 0 | 0 | 1 | 0 | 0 |
| v35 | 0 | 0 | 0 | 0 | 1 | 1 |
| v36 | 0 | 0 | 0 | 0 | 1 | 1 |
| v37 | 0 | 8 | 1 | 1 | 1 | 1 |

Table 4.50 Space fault detection summary

| Subject Programs | Space | |
|---|---|---|
| | Killed | not killed |
| pure-random1 | 12 | 20 |
| pure-random2 | 15 | 17 |
| modeled-random1 | 23 | 9 |
| modeled-random2 | 26 | 6 |
| 2-way | 28 | 4 |
| 3-way | 30 | 2 |

To find out why some faults were not detected, we conducted an investigation to determine the strength of the faults mentioned above. The notion of fault strength or degree of fault is introduced to show the number of parameters that are involved in causing the fault.

Our investigation suggests that the strengths of fault for v12, v18, v27, and v33, are likely to be 4, 5, 7, and 5, respectively. This explains why they were not detected by some approaches. Note that a t-way test set also contains higher strength combinations. This is why v12 and v18 were detected by 3-way testing, even though they have a strength higher than 3. Similarly, the v33 was detected by modeled-random testing.

It is important to note that it can be difficult to determine the strength of fault for a large and/or complex program. In the following, we use v33 as an example to show how we determined the strength of a fault. The test case that killed the fault v33 includes 10 parameters. (Table 4.51)

Table 4.51 Test case parameters of v33

| Test Factors | Test Values |
|---|---|
| Grid | [square, trang, rectang, hex] |
| geometry | [rectangular, circular] |
| Geop | [>0, <0,0] |
| geoQ | [>0, <0,0] |
| polarization | [NA,linear, circular] |
| Add type | [node, block, poly, hex] |
| THETA | [NA,>0, <0,0] |
| PHI | [NA,>0, <0,0] |
| PSI | [NA,>0, <0,0] |
| phase | [NA, uniform, secondorder, rotation, pointing] |

In order to identify the suspicious parameters, we generated 20 more test cases by changing one parameter at a time and fixing the others. 8 out of 20 test cases were able to kill the version (Table 4.52). By comparing the parameter values of these test cases, we were able to detect five suspicious parameters that could cause the fault.

To determine the strength of the fault, we generated 486 exhaustive test cases by fixing the value of the suspicious parameters. We randomly executed 10 out of 486, which they all failed. Therefore, we believe the strength of this fault is likely to be 5. [42]

We performed a similar investigation for v12 and v18 which both were killed only by 3-way testing.

In addition, we performed an investigation for v27. This version was not killed by any of our tests. The code coverage data showed that 14 out of 315 test cases executed the faulty statement. We traced the source code while executing the identified test cases. We applied the same method as described above.

Although a 3-way test set guarantees to kill the faulty version when the fault strength does not go over 3, but it is possible that a 3-way test set kills a version with fault strength greater than 3. Hence 3-way testing was able to kill v12 and v18.

Table 4.52 Additional test cases for v33 to identify fault strength

| Parameters | Grid | Geometry | geoP | geoQ | Polarization | Add | Node orientation | | | phase | result |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | THETA | PHI | PSI | | |
| Original test | Rectang | rectangular | >0 | >0 | linear | node | >0 | >0 | >0 | rotation | fail |
| 1 | Square | rectangular | >0 | >0 | linear | node | >0 | >0 | >0 | rotation | pass |
| 2 | Triang | rectangular | >0 | >0 | linear | node | >0 | >0 | >0 | rotation | pass |
| 3 | Hex | rectangular | >0 | >0 | linear | node | >0 | >0 | >0 | rotation | pass |
| 4 | Rectang | circular | =0 | =0 | linear | node | >0 | >0 | >0 | rotation | pass |
| 5 | Rectang | rectangular | <0 | >0 | linear | node | >0 | >0 | >0 | rotation | fail |
| 6 | Rectang | rectangular | >0 | <0 | linear | node | >0 | >0 | >0 | rotation | fail |
| 7 | Rectang | rectangular | >0 | >0 | circular | node | >0 | >0 | >0 | rotation | fail |
| 8 | Rectang | rectangular | >0 | >0 | NA | node | >0 | >0 | >0 | rotation | fail |
| 9 | Rectang | rectangular | >0 | >0 | linear | block | NA | NA | NA | rotation | pass |
| 10 | Rectang | rectangular | >0 | >0 | linear | poly | NA | NA | NA | rotation | pass |
| 11 | Rectang | rectangular | >0 | >0 | linear | hex | NA | NA | NA | rotation | pass |
| 12 | Rectang | rectangular | >0 | >0 | linear | node | =0 | =0 | =0 | rotation | pass |
| 13 | Rectang | rectangular | >0 | >0 | linear | node | <0 | >0 | >0 | rotation | fail |
| 14 | Rectang | rectangular | >0 | >0 | linear | node | >0 | <0 | >0 | rotation | fail |
| 15 | Rectang | rectangular | >0 | >0 | linear | node | >0 | >0 | <0 | rotation | fail |
| 16 | Rectang | rectangular | >0 | >0 | linear | node | >0 | >0 | >0 | uniform | pass |
| 17 | Rectang | rectangular | >0 | >0 | linear | node | >0 | >0 | >0 | seconorder | pass |
| 18 | Rectang | rectangular | >0 | >0 | linear | node | >0 | >0 | >0 | pointing | pass |
| 19 | Rectang | rectangular | >0 | >0 | linear | node | >0 | >0 | >0 | pqpha | pass |
| 20 | Rectang | rectangular | >0 | >0 | linear | Node | >0 | >0 | >0 | NA | pass |

### *4.5.4 Make*

Make is a tool which controls the generation of executable. Make gets its knowledge of how to build the program from a file called the *makefile*, which lists each of the non-source files and how to compute it from other files. When a program is written, a *makefile* is written for it too. The *makefile* makes it possible to use Make to build and install this program.

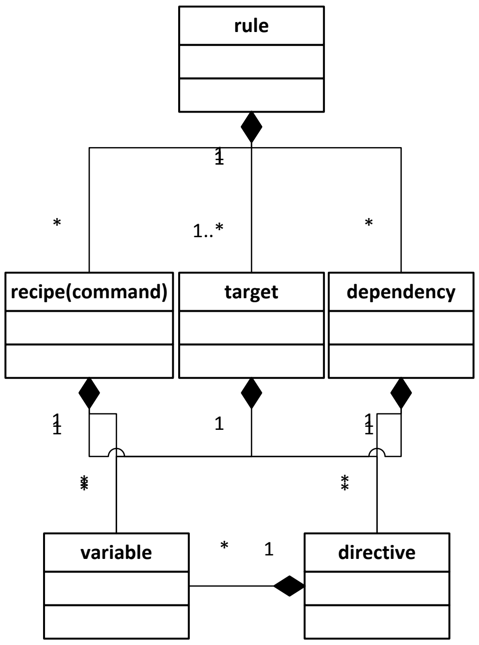


Figure 4.26 Makefile structure model

4.5.4.1 Modeling for the Makefile

We need a file called a makefile to tell make what to do. Makefiles contain four main components: rule, variable definitions, directives, and comments. Figure 4.26 represents the structure of the makefile as a graph model.

4.5.4.1.1 Variable

A variable is a name defined in a makefile to represent a string of text, called the variable's value. These values are substituted into targets, prerequisites, recipes, and other parts of the makefile. A variable name may be any sequence of characters not containing ':', '#', '=', or leading or trailing whitespace. Variable names are case-sensitive. There are some automatic variables that they have particular specialized uses.

Table 4.53 Test factors of variable

| Test Factors | Test Values |
|---|---|
| variable_type | [environment, implicit, override, automatic, special] |
| var_name_uppercase | [begin,middle,end,NA] |
| var_name_number | [begin,middle,end,NA] |
| var_name_underscore | [begin,middle,end.NA] |
| Flavors | [recursively,simply] |
| reference_val | [substitution, nested] |
| variable_value | [pattern, target] |

There are 22 automatic variable exists in make. These variables have values computed again for each rule that is executed, based on the target and prerequisites of the rule. In order to test these variables, when the value of variable_type is equal to special, the concrete test generator will pick one of these automatic variables. The goal is to test each of them at least once.

Table 4.54 List of some of automatic variables

| Variable | Description |
|---|---|
| @ | The file name of the target of the rule |
| % | The target member name, when the target is an archive member |
| < | The name of the first prerequisite |
| ? | The names of all the prerequisites that are newer than the target, with spaces between them |
| ^ | The names of all the prerequisites, with spaces between them |
| + | This is like '$^', but prerequisites listed more than once are duplicated in the order they were listed |
| \| | The names of all the order-only prerequisites, with spaces between them |
| * | The stem with which an implicit rule matches; e.g. If the target is dir/a.foo.b and the target pattern is a.%.b then the stem is dir/foo |

There are some special variables that are used specially by GNU make such as MAKEFILES, VPATH, MAKE and so on. For example the following line specifies a path containing two directories for make to search: `src' and `../headers'.

```
VPATH = src:../headers
```

#### 4.5.4.1.2 Directives

A directive is an instruction for make to do something special while reading the makefile; e.g. include other makefiles, conditional part of makefile, or defining multiline variables.

One occasion for using ‘include’ directives is when several programs, handled by individual makefiles in various directories, need to use a common set of variable definitions or pattern rule.

Table 4.55 Test factors of directive

| Test factors | Test Values |
|---|---|
| Directive_type | [include, condition, define, export, override, vpath] |
| Include_filename | [NA,empty,one,more] |
| Conditional_type | [NA,none, ifeq, ifneq] |
| Define_type | [none, oneline, mulipleline] |
| Export_variable | [=, :=, +=,NA] |
| Override_varibale | [=,:=,+=,NA] |
| Vpath_type | [type1,type2,type3,NA] |

Example of ‘conditional’ directives is when it tells the make to use one set of libraries if the CC variable is ‘gcc’, and a different set of libraries otherwise. It works by controlling which of two recipe lines will be used for the rule.

Another way to set the value of a variable is to use the ‘define’ directive. Also if a variable has been set with a command argument, then ordinary assignments in the makefile are ignored. In order to set a variable in the makefile even though it was set with a command argument, we can use an ‘override’ directive.

The vpath directive specifies a search path for a particular class of file names There are three forms of the vpath directive:

1. Type1: vpath pattern directories: Specify the search path directories for file names that match pattern,
2. Type2: vpath pattern: Clear out the search path associated with pattern.
3. Type3: Vpath: Clear all search paths previously specified with vpath directives.

#### 4.5.4.1.3 Rule

A simple makefile consists of “rule” with the following shape:

```
target ... : prerequisites ...
        recipe
        ...
```

#### 4.5.4.1.4 Target

A target is usually the name of a file that is generated by a program (output file names); examples of targets are executable or object files. A target can also be the name of an action to carry out, such as ‘clean’. The ‘clean’ is a phony target because it is not really the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request.

There are different types of targets: phony, force, empty, special. Special targets such as .PHONY, .SUFFIXES and .DEFAULT are built-in targets. Any defined implicit rule suffix also counts as a special target. In practice, suffixes normally begin with ‘.’, so these special target names also begin with ‘.’.

Also instead of writing many rules, each with one target; we can have one rule with multiple targets. *Static pattern rules* apply to multiple targets and can vary the prerequisites according to the target name.

Table 4.56 Test factors of target

| Test factors | Test Values |
|---|---|
| special_target | [phony, suffixes, default, precious, intermediate, secondary, secondary expansion, ignore, silent, notparallel] |
| target_type | [phony, force, empty, special] |
| targetname period | [begin, mid, end, NA] |
| targetname underscores | [begin, mid, end, NA] |
| targetname Digits | [begin, mid, end, NA] |
| directive | [on, off] |
| variable | [on, off] |

When the value of target_type is equal to ‘force’ then the target of the rule is a nonexistent file.

#### 4.5.4.1.5 Prerequisite

A prerequisite is a file that is used as input to create the target. A target often depends on several files (input file names). There are actually two different types of prerequisites: normal and order-only.

If the value of search_dir is 'general', then the value of VPATH variable specifies a list of directories that make should search. Otherwise if the value is 'selective', the value of vpath directive specifies a search path for a particular class of file names.

Table 4.57 Test factors of prerequisite

| Test factors | Test Values |
|---|---|
| Type | [normal, order-only] |
| search_dir | [general, selective, NA] |
| Variable | [on,off] |
| Directive | [on, off] |

#### 4.5.4.1.6 Recipe

A recipe is an action to make. A recipe may have more than one command. Usually a recipe is in a rule with prerequisites and serves to create a target file if any of the prerequisites change. However, for example the rule containing the delete command associated with the target 'clean' does not have prerequisites. There are different types of recipe: canned and empty. When the value of type is 'canned' then the same sequence of commands is used in making various targets. Otherwise, when it is 'empty' no target will get implicit recipes from implicit rules.

Make knows how to execute several recipes at once. So when the parallel test factor is on then several recipes will execute at once.

Table 4.58 Test factors of recipe

| Test factors | Test Valus |
|---|---|
| Type | [canned, empty] |
| Define directive | [on, off]] |
| Variable | [on,off] |
| Echoing | [on, off] |
| Parallel | [on, off] |
| Recursive | [on, off] |

Implicit rules are certain standard ways of remaking target files such as either write a rule with no recipe, or don't write a rule at all. Then make will figure out which implicit rule to use based on which kind of source file exists or can be made. You can define your own implicit rules by writing *pattern rules. Suffix rules* are a more limited way to define implicit rules. Pattern rules are more general and clearer, but suffix rules are retained for compatibility.

Suffix rules are the old-fashioned way of defining implicit rules for make. Suffix rules are obsolete because *pattern rules* are more general and clearer. Therefore, we didn't model these rules.

Double-colon rules are explicit rules written with '::' instead of ':' after the target names. Each double-colon rule should specify a recipe; if it does not, an implicit rule will be used if one applies.

Static pattern rules are rules which specify multiple targets and construct the prerequisite names for each target based on the target name. The static pattern rule can be better than an *implicit rule*.

Every operating system such as VMS, Windows, and OS/2 has different sets of default rules. Also the recipes in built-in implicit rules make liberal use of certain predefined variables. Since there is a long list of variables we just modeled some of the most common one as are shown in

Table 4.60.

Table 4.59 Test factors of rule

| Test factors | Test Values |
|---|---|
| rule_type | [explicit, implicit] |
| Implicit_rule | [off, suffix, pattern, Last-Resort] |
| Explicit_rule | [off, colon, double-colon ] |
| Target | [one, two or more] |
| Prerequisite | [zero, one, more] |
| Recipe | [zero, one, more] |
| whildcard extension | [on, off] |
| shorthand | [on, off] |
| File_changed | [none, one or more not all, all] |

Table 4.60 Some of the common implicit rules

| Variable | Description |
|---|---|
| CC | Program for compiling C programs |
| CPP | Program for running the C preprocessor, with results to standard output |
| RM | Command to remove a file |
| CFLAGS | Extra flags to give to the C compiler. |
| CPPFLAGS | Extra flags to give to the C preprocessor |

Last-Resort Default Rules are implicit rules with no prerequisites. If a rule has no prerequisites (prerequisites parameter is off) or recipe (recipe parameter is off), and the target of the rule is a nonexistent file, then make imagines this target to have been updated whenever its rule is run. This implies that all targets depending on this one will always have their recipe run.

We used conventions for writing the Makefiles. For example all of our makefiles contain this line:

```
SHELL = /bin/sh
```

Moreover we set the suffix list explicitly using only the suffixes that are needed in the particular Makefile, like this:

```
.SUFFIXES:
.SUFFIXES: .c .o
```

In addition, we need to model the number of c files that are going to change each time before the test cases executed. Therefore, we identified the parameter 'file_changed' with three values [none, one or more not all, all].

4.5.4.2 Modeling for the Command Line Interface

Table 4.61 Command line test factors

| Test Factors | Test Values | Description |
|---|---|---|
| -C dir | [on,off] | Change to directory *dir* before reading the makefiles. |
| -d | [on,off] | Print debugging information in addition to normal processing |
| -e | [on,off] | Give variables taken from the environment precedence over variables from makefiles |
| -f file | [on,off] | Read the file named *file* as a makefile. |
| -i | [on,off] | Ignore all errors in recipes executed to remake files |
| -I dir | [on,off] | Specifies a directory *dir* to search for included makefiles |
| -j job | [on,off] | Specifies the number of recipes (jobs) to run simultaneously |
| -l load | [on,off] | Specifies that no new recipes should be started if there are other recipes running |
| -k | [on,off] | Continue as much as possible after an error |
| -n | [on,off] | Print the recipe that would be executed, but do not execute it |
| -o file | [Alone, off] | Do not remake the file *file* even if it is older than its prerequisites |
| -p | [on,off, With –f/dev/null] | Print the rules and variable values that results from reading the makefiles; then execute as usual or as otherwise specified |
| -q | [on,off] | just return an exit status that is zero if the specified targets are already up to date |
| -r | [on,off] | Eliminate use of the implicit rules |
| -s | [on,off] | Silent operation; do not print the recipes as they are executed |
| -S | [on,off] | Cancel the effect of the '-k' option. |
| -t | [on,off] | Touch files instead of running their recipes |
| -v | [on,off, With –f/dev/null]] | Print the version of the make program plus a copyright, a list of authors, and a notice that there is no warranty; then exit |
| -w | [on,off] | Print a message containing the working directory both before and after executing the makefile |
| -W file | [on,off] | Pretend that the target *file* has just been modified |

Table 4.62 Summary of make input model

| Application Space | # of parameter values | # of parameter | # of constraint parameters | # of constraint | # of relation parameters | # of relation |
|---|---|---|---|---|---|---|
| Variable | $2^3 4^3 5^1$ | 7 | $3^3$ | 3 | $3^1$ | 1 |
| Directive | $2^3 3^2$ | 5 | - | - | - | - |
| Target | $4^4 10^1$ | 5 | - | - | - | - |
| Prerequisite | $2^2 3^1 4^1$ | 4 | - | - | - | - |
| Recipe | $2^6$ | 6 | - | - | - | - |
| Rule | $2^4 3^4 4^1$ | 9 | - | - | - | - |
| CMD | $2^{18} 3^2$ | 20 | - | - | - | - |
| TOTAL | $2^{36} 3^9 4^9 5^1 10^1$ | 56 | $3^3$ | 3 | $3^1$ | 1 |

4.5.4.3 Results and discussion

The designed model for makefile has 7 IPMs which are shown in Table 4.62, yielding 1230 generated test cases. Code coverage data are shown in Figure 4.27. We used *Gcov* to collect code coverage. Make statistics are shown in Table 4.63. e.g. number of lines of code in Make are 35545.

Table 4.63 Make statistics

| | |
|---|---|
| LOC (line of code) | 35,545 |
| Number of Branches | 4538 |
| Number of Functions | 268 |
| Number of faulty versions | 19 |
| Type of Faults | Seeded |

Gcov gave us the code coverage for all of the test cases. While we executed our tests, it marked the parts of the source code that were executed. It is shown that our tests covered almost 80% of lines.

As it is shown in Figure 4.27, the code coverage of modeled-random1 and modeled-random2 is the same. In addition, in this subject we notice the code coverage of 2-way testing is less than modeled-random1. Therefore, we conducted an investigation for this subject. We generated nine more random test-set for modeled-random1 (total 10 test sets) and nine additional random test-set for modeled-random2. The results of the code coverage are shown in a sorted-order in Table 4.64. The average code coverage of the modeled-random1 is 61%

which is less than code coverage of 2-way testing (66%). This is also true for the code coverage of average modeled-random2 and 3-way testing. This suggests that, the t-way testing have a better code coverage than the average of several modeled-random set of the same size.

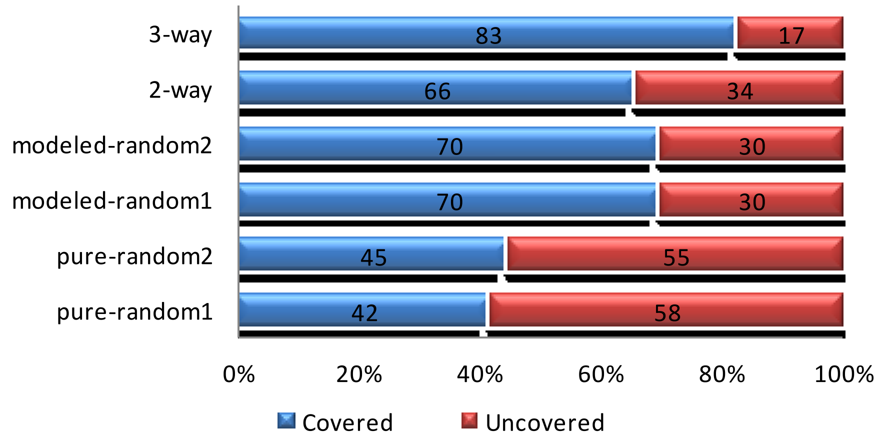


Figure 4.27 Make code coverage results

Another observation from the Table 4.64 suggests that, modeled-random1 cannot perform better than modeled-random2. In the best case scenario, modeled-random1 is equivalent to the worst case scenario of modeled-random2.

Table 4.64 Code coverage results

| Test sets# | Modeled-random1 | | Modeled-random2 | |
|---|---|---|---|---|
| | Branch coverage (%) | Statement coverage (%) | Branch coverage (%) | Statement coverage (%) |
| 1 | 44 | 54 | 62 | 70 |
| 2 | 44 | 55 | 65 | 72 |
| 3 | 46 | 56 | 69 | 76 |
| 4 | 50 | 59 | 71 | 79 |
| 5 | 50 | 59 | 72 | 80 |
| 6 | 50 | 60 | 74 | 80 |
| 7 | 53 | 62 | 77 | 83 |
| 8 | 54 | 63 | 78 | 84 |
| 9 | 59 | 68 | 78 | 84 |
| 10 | 62 | 70 | 78 | 84 |
| **Avg.** | **51** | **61** | **73** | **79** |
| **t-way** | **57** | **66** | **77** | **83** |

The Figure 4.28 shows the same results. The blue line and red line represent the modeled-random1 and modeled-random2 respectively for ten modeled-random test sets.

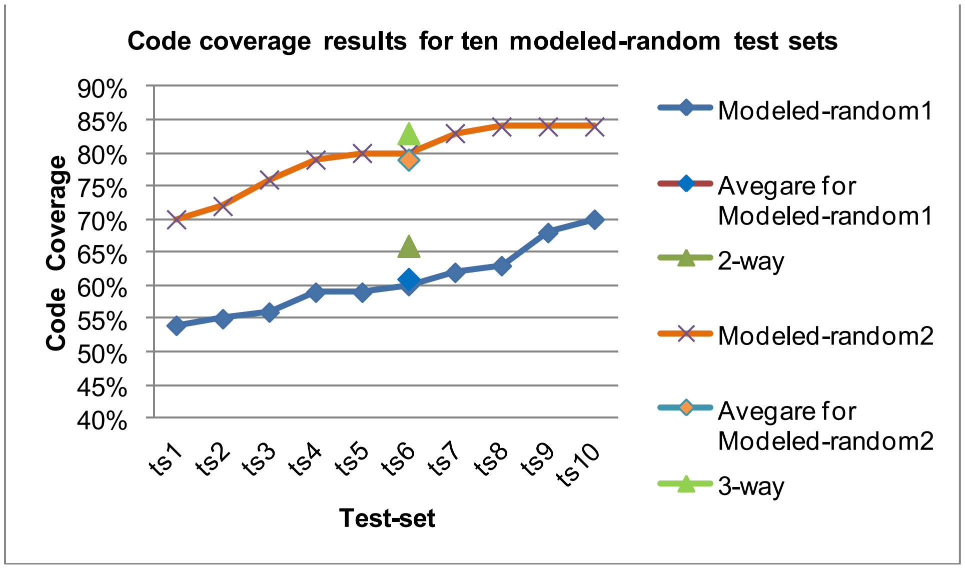


Figure 4.28 Code Coverage results

In addition, we investigate to find out how many t-way combinations are covered by a modeled-random test set. The results are shown in Table 4.65 suggest that with the same number of test cases as t-way testing, modeled-random on average covers 87.5% of t-way combinations.

Table 4.65 T-way coverage results

| | Modeled-random1 | Modeled-random2 |
|---|---|---|
| | 2-way coverage (%) | 3-way coverage (%) |
| 1 | 79 | 87 |
| 2 | 81 | 89 |
| 3 | 83 | 89 |
| 4 | 83 | 89 |
| 5 | 84 | 91 |
| 6 | 86 | 91 |
| 7 | 86 | 91 |
| 8 | 87 | 92 |
| 9 | 88 | 92 |
| 10 | 88 | 92 |
| **Avg.** | **85** | **90** |

The Figure 4.29 shows the same results. The blue line and red line represent the 2-way coverage of modeled-random1 and 3-way coverage of modeled-random2 respectively for ten modeled-random test sets and the dash-lines show their average.

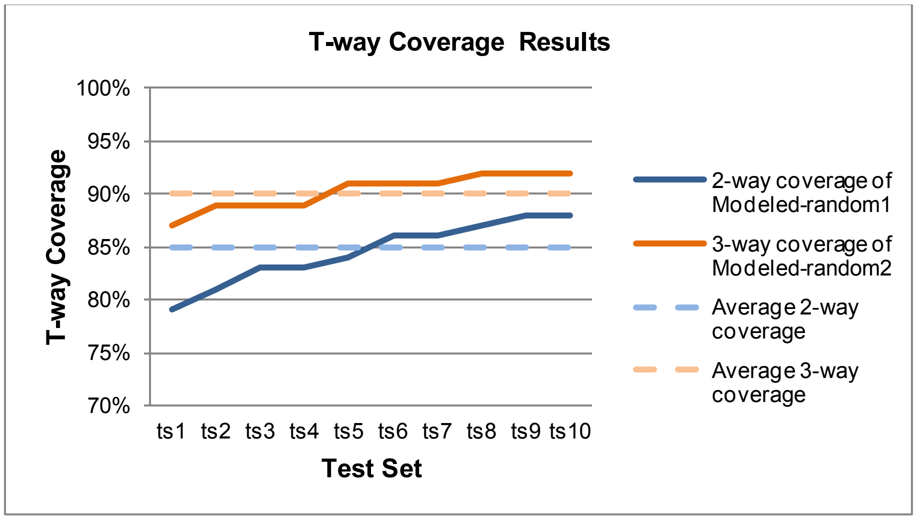


Figure 4.29 T-way Coverage Results

The 2-way testing miss more faults. We notice almost the same result with t=3. In this subject we notice that the 3-way testing has better fault detection than modeled-random2. The total number of detected faults with 3-way testing is only 9 out of 19.

Table 4.66 Make fault detection results

| Subject Programs | Make | |
|---|---|---|
| | Killed | not killed |
| pure-random1 | 2 | 17 |
| pure-random2 | 2 | 17 |
| modeled-random1 | 6 | 13 |
| modeled-random2 | 6 | 13 |
| 2-way | 6 | 13 |
| 3-way | 9 | 10 |
| 4-way | 14 | 5 |

In order to further investigate the results we increased the strength of t. The fault detection of 4-way testing improved. This suggests that the strengths of faults are likely to be higher than 3-way test set. This explains why they were not detected by some approaches. Note that a t-way test set also contains higher strength combinations. We got 89% code coverage for 4-way testing with 4503 test cases. As the number of test cases increases rapidly as the test strength increases; therefore, we did not go beyond 4-way testing.

### *4.5.5 Grep*

Grep, from GNU website [18], is a program to search for strings inside an *input file*. It searches input file for lines containing a match to a given *pattern* list. When it finds a match in a line, it produces the output. We downloaded two consecutive versions of Grep 2.5.3 and 2.5.4. The newer version is a 'fixed bug only' version; therefore, no additional features added to the system. We considered the older version as our faulty version and the newer version as our clean version. The faulty version contains five bugs. We generated five faulty versions; each contains one of the bugs.

#### 4.5.5.1 Modeling for the Pattern

The major variants of Grep patterns, controlled by the following options [18]:

- "G" Interpret the pattern as a basic regular expression (BRE) which is the default.
- "E" Interpret the pattern as an extended regular expression (ERE).
- "F" Interpret the pattern as a list of fixed strings, separated by newlines, any of which is to be matched.
- "P" Interpret the pattern as a Perl regular expression.

Based on different meta-characters in a pattern, the program may behave differently. In addition, we can categorize these meta-characters to six different groups based on their characteristics as it is shown in Table 4.67. Therefore, we must cover all the groups of the meta-characters in order to cover all behaviors of the subject program.

A regular expression may be followed by one of several repetition operators. Most characters, including all letters and digits, are regular expressions that match themselves. The letter and digit parameters in Table 4.68 are identified to cover these types of regular expression. In addition, any meta-character with special meaning must be used by preceding a backslash. The meta-char parameter corresponds to this type. The bracket and special expression are also regular expressions [18].

Table 4.67 Regular expression

| Group | | Meta-character | Descriptions |
|---|---|---|---|
| Reputation operators | 1 | . | The period '.' matches any single character |
| | 2 | ? | The preceding item is optional and will be matched at most once |
| | 3 | * | The preceding item will be matched zero or more times |
| | 4 | + | The preceding item will be matched one or more times |
| | 5 | {n} | The preceding item is matched exactly n times |
| | 6 | {n,m} | The preceding item is matched at least n times and at most m times |
| | 7 | {n,} | The preceding item is matched at least n times |
| | 8 | {,m} | The preceding item is at most m times |
| Infix operator | | \| | The '\|' joins two regular expressions |
| Bracket expression | | [*List*] | The enclosed *List is a* list of characters. It matches any single character in that list |
| | | [^*List*] | If the first character of the list is the caret '^', then it matches any character not in the list |
| | | [-] | Two characters separated by a hyphen. It matches any single character that sorts between the two characters |
| | 1 | [alnum] | Match non-word constituent, (\w) , this is the same as '[0-9A-Za-z]' |
| | 2 | [alpha] | Alphabetic characters, lower and upper |
| | 3 | [blank] | Blank characters: space and tab |
| | 4 | [digit] | Digits: 0 1 2 3 4 5 6 7 8 9 |
| | 5 | [graph] | Graphical characters: '[:alnum:]' and '[:punct:]'. |
| | 6 | [lower] | Lower-case letters |
| | 7 | [print] | Printable characters: '[:alnum:]', '[:punct:]', and space. |
| | 8 | [punct] | Punctuation characters; ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ' { \| } ~ |
| | 9 | [upper] | Upper-case letters |
| | 10 | [xdigit] | Hexadecimal digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f |
| Special expression | 1 | \b | Match the empty string at the edge of a word |
| | 2 | \B | Match the empty string provided it's not at the edge of a word |
| | 3 | \< | Match the empty string at the beginning of word |
| | 4 | \> | Match the empty string at the end of word |
| | 5 | \w | Match word constituent |
| | 6 | \W | Match non-word constituent |
| | 7 | \s | Match whitespace |
| | 8 | \S | Match non-whitespace |
| Anchoring | | ^ | Match the empty string at the beginning of a line |
| | | $ | Match the empty string at the end of a line |
| Sub-expression | | \n | n is a single digit, matches the substring previously matched by the nth parenthesized sub-expression of the regular expression |

Table 4.68 Test factors of individual expression

| Test factor | Test value |
|---|---|
| bracket_predef | [alnum, alpha, digit, graph, lower, print, punct, space, upper, xdigit,off] |
| bracket_negate | [on, off] |
| bracket_range | [on, off] |
| bracket_pos | [off, begin, middle, end] |
| bracket_repeat | [off,., ?, * ,+ ,{n,m} ,{n} ,{n,} ,{,m}] |
| letter_pos | [off, begin, middle, end] |
| letter_repeat | [off,., ?, * ,+ ,{n,m} ,{n} ,{n,} ,{,m}] |
| meta-char_pos | [off, begin, middle, end] |
| meta-char_repeat | [off,., ?, * ,+ ,{n,m} ,{n} ,{n,} ,{,m}] |
| digit_pos | [off, begin, middle, end] |
| digit_repeat | [off,., ?, * ,+ ,{n,m} ,{n} ,{n,} ,{,m}] |
| Special | [off, \b, \B, \w, \W, \s, \S, \<, \>] |
| special_pos | [off, begin, middle, end] |
| special_repeat | [off,., ?, * ,+ ,{n,m} ,{n} ,{n,} ,{,m}] |
| EOL | [on, off] |
| BOL | [on, off] |

To cover the repetition operators we have two approaches. One approach is that, considering two parameters for repetition operators, one of them shows the preceding regular expression and other indicates the position of repetition operators.

Another approach is that, combining repetition operators and its regular expression to create new parameters, e.g. the bracket_repeat parameter checks the behavior of the system when the previous element of the repetition operator is a bracket expression.

We select the second approach to cover the repetition operator for each mentioned regular expression (bracket, letter, digit, special and meta-char). A parameter that its name ends with "_repeat" is introduced for each regular expression. The reason is that since the repetition operators are complex, it is very likely that the program has bug relates to them. Therefore, we can have more complex pattern with multiple repeat parameters for each regular expression.

In addition, some meta-character such as EOL and BOL has fixed position in the pattern, they can just appear at the beginning or end of the pattern, but some expressions such as bracket expression can appear at the different positions in the pattern, beginning, middle or end. So another parameter bracket_pos is identified to cover different positions. All the parameters that their names end with "_pos" are corresponding to the position of the abstract parameter in a pattern. For example, in Table 4.68, the letter_pos parameter corresponding to letter regular expression that is a letter, if the value of letter_pos is 'begin', this means that the expression will begin with a letter. If the value of letter_repeat is a '*', then this means that after the letter, we have a '*' meta-character in the pattern 'a*'.

It is assumed that the relative positions of elements do not make any change in testing phase, e.g. two parameters can have 'middle' as their values in one test. We can have two concrete tests based on their relative positions. But since relative position does not consider in the abstract model, it is assumed that relative position does not affect the behavior of the system and consequently the test results.

Moreover, we do not consider the number of times that a meta-character is appeared. We assume that if the program is correct for one, it corrects for more than one.

A pattern consists of one or more expressions. Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated expressions. Two regular expressions may be joined by the infix operator '|'; the resulting regular expression matches any string matching either alternate expression. A whole expression may be enclosed in parentheses form a sub-expression [18].

Table 4.69 Test Factors of multiple expressions

| Test factor | Test value |
|---|---|
| concatenate | [on, off] |
| alternate | [on, off] |
| Num_of_expression | [one, two, more] |
| Backrefrence | [on, off] |

The back-reference '\n', where n is a single digit, matches the substring previously matched by the nth parenthesized sub-expression of the regular expression. For example, '(a)\1' matches 'aa'.

When used with alternation, if the group does not participate in the match then the back-reference makes the whole match fail. For example, ‘a(.)|b\1’ will not match ‘ba’ [18].

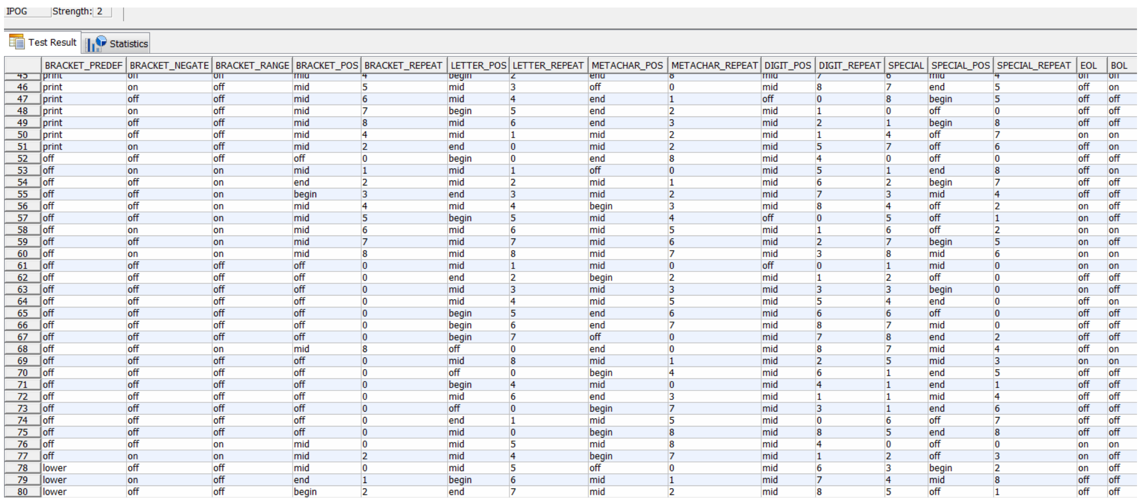
To model a pattern with two or more expressions, first we model an individual expression and then we model multiple expressions to form a pattern as it is shown in Table 4.69. We cannot use the *concatenate* and *alternate* operations if the pattern has only one expression; therefore, the value of these two parameters (*concatenate* and *alternate*) set to off. At this point, all needed information for generating patterns is available. As no human decision is needed, the concrete tests are generated automatically.

### 4.5.5.2 Modeling for the Input file

The second input parameter of the Grep program is the input file. The behavior of Grep highly depends on it. Consider the fault is revealed only when the pattern is matched. If the input file does not contain a match pattern, the fault does not reveal. We do not identify any parameter for input file in the model, but we create the input file based on abstract parameters of the pattern model in a way that it covers possible situations.

The relation between an abstract test and its corresponding concrete tests is one to many, i.e. one abstract test can map to more than one concrete tests. The derivation part is done automatically, and one concrete test is selected.

Pattern is built based on their properties in the abstract test. For the input file, we create a file which contains a line matched the pattern only once and the other line that matched the pattern twice. Thus, we test the situation that the pattern appears in the file or on the target line more than once. Then we select the number of lines such that in each line just one element in the pattern does not match with the pattern. At the end, one line is added to violate all elements in the pattern. The input file is completed and covers all texts that potentially could make error.



IPOG Strength: 2

Test Result | Statistics

| | BRACKET_PREDEF | BRACKET_NEGATE | BRACKET_RANGE | BRACKET_POS | BRACKET_REPEAT | LETTER_POS | LETTER_REPEAT | METACHAR_POS | METACHAR_REPEAT | DIGIT_POS | DIGIT_REPEAT | SPECIAL | SPECIAL_POS | SPECIAL_REPEAT | EOL | BOL |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 45 | print | on | on | mid | 4 | begin | 2 | end | 8 | mid | 7 | 6 | mid | 4 | on | on |
| 46 | print | on | off | mid | 5 | mid | 3 | off | 0 | mid | 8 | 7 | end | 5 | off | on |
| 47 | print | off | off | mid | 6 | mid | 4 | end | 1 | off | 0 | 8 | begin | 5 | off | off |
| 48 | print | on | off | mid | 7 | begin | 5 | end | 2 | mid | 1 | 0 | off | 0 | off | off |
| 49 | print | off | off | mid | 8 | mid | 6 | end | 3 | mid | 2 | 1 | begin | 8 | off | off |
| 50 | print | off | off | mid | 4 | mid | 1 | mid | 2 | mid | 1 | 4 | off | 7 | on | on |
| 51 | print | on | off | mid | 2 | end | 0 | mid | 2 | mid | 5 | 7 | off | 6 | off | on |
| 52 | off | off | off | off | 0 | begin | 0 | end | 8 | mid | 4 | 0 | off | 0 | off | off |
| 53 | off | on | on | mid | 1 | mid | 1 | off | 0 | mid | 5 | 1 | end | 8 | off | on |
| 54 | off | off | on | end | 2 | mid | 2 | mid | 1 | mid | 6 | 2 | begin | 7 | off | off |
| 55 | off | off | on | begin | 3 | end | 3 | mid | 2 | mid | 7 | 3 | mid | 4 | off | off |
| 56 | off | off | on | mid | 4 | mid | 4 | begin | 3 | mid | 8 | 4 | off | 2 | on | off |
| 57 | off | off | on | mid | 5 | begin | 5 | mid | 4 | off | 0 | 5 | off | 1 | on | off |
| 58 | off | on | on | mid | 6 | mid | 6 | mid | 5 | mid | 1 | 6 | off | 2 | on | on |
| 59 | off | off | on | mid | 7 | mid | 7 | mid | 6 | mid | 2 | 7 | begin | 5 | on | off |
| 60 | off | on | on | mid | 8 | mid | 8 | mid | 7 | mid | 3 | 8 | mid | 6 | on | on |
| 61 | off | off | off | off | 0 | mid | 1 | mid | 0 | off | 0 | 1 | mid | 0 | on | on |
| 62 | off | off | off | off | 0 | end | 2 | begin | 2 | mid | 1 | 2 | off | 0 | off | off |
| 63 | off | off | off | off | 0 | mid | 3 | mid | 3 | mid | 3 | 3 | begin | 0 | on | off |
| 64 | off | off | off | off | 0 | mid | 4 | mid | 5 | mid | 5 | 4 | end | 0 | off | on |
| 65 | off | off | off | off | 0 | begin | 5 | end | 6 | mid | 6 | 6 | off | 0 | off | off |
| 66 | off | off | off | off | 0 | begin | 6 | end | 7 | mid | 8 | 7 | mid | 0 | off | off |
| 67 | off | off | off | off | 0 | begin | 7 | off | 0 | mid | 7 | 8 | end | 2 | off | off |
| 68 | off | off | on | mid | 8 | off | 0 | end | 0 | mid | 8 | 7 | mid | 4 | off | on |
| 69 | off | off | off | off | 0 | mid | 8 | mid | 1 | mid | 2 | 5 | mid | 3 | on | on |
| 70 | off | off | off | off | 0 | off | 0 | begin | 4 | mid | 6 | 1 | end | 5 | off | off |
| 71 | off | off | off | off | 0 | begin | 4 | mid | 0 | mid | 4 | 1 | end | 1 | off | off |
| 72 | off | off | off | off | 0 | mid | 6 | end | 3 | mid | 1 | 1 | mid | 4 | off | off |
| 73 | off | off | off | off | 0 | off | 0 | begin | 7 | mid | 3 | 1 | end | 6 | off | off |
| 74 | off | off | off | off | 0 | end | 1 | mid | 5 | mid | 0 | 6 | off | 7 | off | off |
| 75 | off | off | off | off | 0 | mid | 0 | begin | 8 | mid | 8 | 5 | end | 8 | off | off |
| 76 | off | off | on | mid | 0 | mid | 5 | mid | 8 | mid | 4 | 0 | off | 0 | on | on |
| 77 | off | on | on | mid | 2 | mid | 4 | begin | 7 | mid | 1 | 2 | off | 3 | on | off |
| 78 | lower | off | off | mid | 0 | mid | 5 | off | 0 | mid | 6 | 3 | begin | 2 | on | off |
| 79 | lower | on | off | end | 1 | begin | 6 | mid | 1 | mid | 7 | 4 | mid | 8 | off | off |
| 80 | lower | off | off | begin | 2 | end | 7 | mid | 2 | mid | 8 | 5 | off | 1 | off | off |

Figure 4.30 Part of grep 2-way test cases

Following is a sample concrete test case for the test case # 52.

| | | |
|---|---|---|
| bracket_predef=Off | letter_repeat=Off | special_pos=Off |
| bracket_negate=On | meta-char_pos=End | special_repeat=Off |
| bracket_range=Off | meta-char_repeat={n,m} | EOL=Off |
| bracket_pos=Off | digit_pos=Mid | BOL=Off |
| bracket_repeat=Off | digit_repeat=+ | |
| letter_pos=Begin | Special=Off | |

Pattern: a1+\+{1,2}

Input file:

1. a1+ (match once)
2. a111+++a1++ (match twice)
3. b111++
4. a22++
5. a++
6. a1
7. a1*
8. b22222*********

4.5.5.3 Modeling for the Command Line Interface

Based on specification [18], there are seven groups of options available in Grep as it is shown in Table 4.70. Some of these options e.g. help, version are totally independent from each other. Because there is no interaction between them they must appear in the test only once. But others such as output line prefixing control options; they are affecting each other. When several prefix fields need to be generated, the order is always file name, line number, and byte offset regardless of the order in which these options were specified [18]. The '-H' option indicates to print the file name for each match and '-h' suppress printing file name on the output. If we use these two options at the same time, they should overwrite each other. In order to test different order between the two options, we identified a new parameter '-h_first'. If in a test, the values of these parameter (-h, -H, and –h_first) are 'on' then we first add –h to the options list and then –H and our expected output result is having no file name in the output.

Also there are some other options such as –q and –s that we don't want to try all different combinations of them with the other options because when their value is 'on' then nothing will be written in the output.

Table 4.70 Command line model

| Group | Test Factors | Test Values | Description |
|---|---|---|---|
| General info | --help | [on,off] | Print briefly summarizing the command-line options |
| | --version | [on,off] | Print the version number of Grep |
| Matching control | -e *pattern* | [on,off] | Use *pattern* as the pattern |
| | -f *file* | [on,off] | Obtain patterns from *file*, one per line |
| | -i | [on,off] | Ignore case distinctions in both the pattern and the input files |
| | -v | [on,off] | Invert the sense of matching, to select non-matching lines |
| | -w | [on,off] | Select only those lines containing matches that form whole words |
| | -x | [on,off] | Select only those matches that exactly match the whole line |
| | -x_first | [on,off] | |
| General output control | -c | [on,off] | Print a count of matching lines |
| | --color | [off, never, always, auto] | Surround the matched (non-empty) strings to display them in color |
| | -L | [on,off] | Print the name of **input file** without match |
| | -l | [on,off] | Print the name of **input file** with matches |
| | -l_first | [on, off] | |
| | -m num | [off, one,more] | Stop reading a file after num matching lines |
| | -o | [on,off] | Print only the matched parts of lines |
| | -q | [on,off] | Don't write anything |
| | -s | [on,off] | No message |
| Output line prefix control | -H | [on,off] | Suppress the prefixing filename on output |
| | -h | [on,off] | Print the filename for each match |
| | -h_first | [yes,no] | |
| | -n | [on,off] | Line number |
| Context line control | -A num | [on,off] | Print NUM lines of leading **context** |
| | -B num | [on,off] | Print NUM lines of trailing **context** |
| | -C num | [on,off] | Print NUM lines of output **context** |
| File and directory selection | -a | [on,off] | Process a binary file as if it were text |
| | -D action | [off,read,skip] | Use action to process device |
| | -d action | [off,read,recurse,skip] | Use action to process directory |
| | -I | [on,off] | Process a binary file as if it did not contain matching data |
| | -r | [on,off] | Read and process all files in that directory recursively |
| | -R | [on,off] | Following all symbolic links recursively following all symbolic links |
| | -z | [on,off] | Treat the input as a set of lines, each terminated by a zero byte |

Table 4.71 Summary of grep input model

| Application Space | # of parameter values | # of parameter | # of constraint parameters | # of constraint | # of relation parameters | # of relation |
|---|---|---|---|---|---|---|
| Expression | $2^4 4^5 9^6 11^1$ | 16 | $2^5 3^3 4^1 6^{13}$ | 22 | $2^2 3^1 5^1$ | 4 |
| CMD | $2^{28} 3^2 4^2$ | 32 | $3^3$ | 3 | $2^3 3^1$ | 4 |
| TOTAL | $2^{32} 3^2 4^7 9^6 11^1$ | 48 | $2^5 3^6 4^1 6^{13}$ | 25 | $2^5 3^2 5^1$ | 8 |

4.5.5.4 Results and discussion

The designed model for Grep has two IPMs which are shown in Table 4.71, yielding 910 generated test cases. Code coverage data are shown in Figure 4.31. We used *Gcov* to collect code coverage. Grep statistics are shown in Table 4.72. e.g. number of lines of code in Grep are 10068.

Table 4.72 Grep statistics

| | |
|---|---|
| LOC (line of code) | 10,068 |
| Number of Branches | 2,723 |
| Number of Functions | 146 |
| Number of faulty versions | 5 |
| Type of Faults | Real |

Gcov gives us the code coverage for all of the test cases. While we execute our tests, it marks the parts of the source code that are executed. It is shown in Figure 4.31 that t-way test set covered 80% of lines. The total number of detected faults is 4 out of 5. All the test sets detect the same faults. This suggests that the strength of the faults for this subject is most likely less than 3.

Table 4.73 Grep fault detection results

| Subject Programs | Grep | |
|---|---|---|
| | killed | not killed |
| modeled-random1 | 4 | 1 |
| modeled-random2 | 4 | 1 |
| 2-way | 4 | 1 |
| 3-way | 4 | 1 |

We notice from the code coverage results for this subject that the modeled_random2 and 3-way testing have the same code coverage. To evaluate our results we generate up to 10 modeled-random test sets.

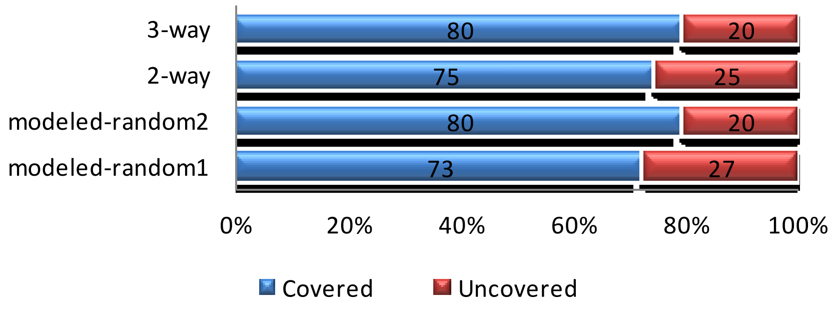


Figure 4.31 Grep code coverage results

The sorted results (Table 4.74) suggest that the average code coverage (branch coverage and statement coverage) for modeled-random is less than t-way testing. The results are identical to the results of the make subject.

Table 4.74 Code coverage results

| Tests# | Modeled-random1 | | Modeled-random2 | |
|---|---|---|---|---|
| | Branch coverage (%) | Statement coverage (%) | Branch coverage (%) | Statement coverage (%) |
| 1 | 59 | 71 | 69 | 77 |
| 2 | 59 | 72 | 70 | 77 |
| 3 | 60 | 72 | 70 | 78 |
| 4 | 61 | 72 | 74 | 78 |
| 5 | 61 | 73 | 74 | 78 |
| 6 | 61 | 73 | 74 | 79 |
| 7 | 61 | 73 | 75 | 79 |
| 8 | 64 | 75 | 77 | 80 |
| 9 | 66 | 75 | 77 | 80 |
| 10 | 68 | 77 | 79 | 82 |
| Ave. | 62 | 73 | 74 | 79 |
| t-way | 64 | 75 | 75 | 80 |

The Figure 4.32 shows the same results. The blue line and red line represent the modeled-random1 and modeled-random2 respectively for ten modeled-random test sets.

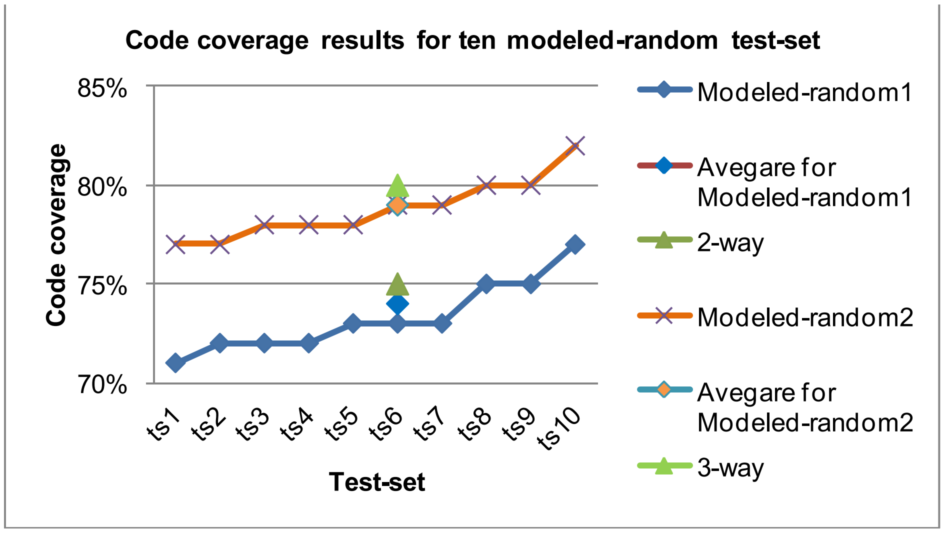


Figure 4.32 Code coverage results

In addition, we investigate to find out how many t-way combinations are covered by a modeled-random test set. The sorted results which are shown in Table 4.76 suggest that with the same number of test cases as t-way testing, modeled-random on average covers 85% of t-way combinations.

Table 4.75 T-way coverage results

| | Modeled-random1 | Modeled-random2 |
|---|---|---|
| | 2-way coverage (%) | 3-way coverage (%) |
| 1 | 79 | 85 |
| 2 | 79 | 86 |
| 3 | 80 | 87 |
| 4 | 80 | 87 |
| 5 | 80 | 88 |
| 6 | 83 | 88 |
| 7 | 83 | 89 |
| 8 | 84 | 89 |
| 9 | 85 | 89 |
| 10 | 85 | 90 |
| Ave. | 82 | 88 |

The Figure 4.33 shows the same results. The blue line and red line represent the 2-way coverage of modeled-random1 and 3-way coverage of modeled-random2 respectively for ten modeled-random test sets and the dash-lines show the average.

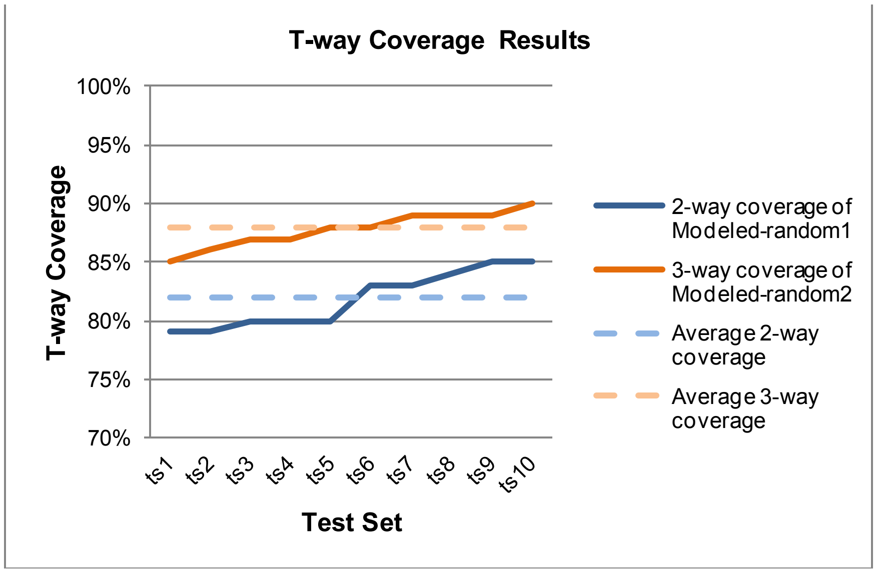


Figure 4.33 T-way coverage results

## 4.6 Summary

For the pure-random approach, we first tried to generate random XML files using *Oxygen* solely based on the DTD file, i.e., without supplying any additional information. This approach only achieved 22% statement coverage on average. Since this approach is so ineffective, we do not consider it in the rest of our case studies.

To make the pure-random approach more meaningful, we provided additional information to the random XML files generation process. There are two types of additional information: (1) Information about the environment such as directory name, file name, class path, etc.; and (2) Constraints that may exist between different elements, e.g. uniqueness constraints, cross-reference constraints, etc.

For example the following is part of the original *Apache Ant* DTD file:

```
<!ELEMENT PROJECT (TARGET)+>
<!ATTLIST PROJECT
   Name    CDATA #IMPLIED
Basedir CDATA #REQUIRED >
```

The PROJECT element has two attributes *Name* and *Basedir* listed in ATTLIST. The CDATA element indicates that the value is a character data. The REQUIRED or IMPLIED element indicates that the value is required or not.

The Name attribute represents the name of the project, which is an optional value. The Basedir attribute represents the base directory of the project. The Basedir attribute is required and cannot be a random string. This is because the *Apache Ant* program terminates if the base directory does not exist. Therefore, we modified the DTD file and fixed the value of Basedir to current directory.

The pure-random approach with such semantic information added to the schema achieved 45% statement coverage on average (Figure 4.34). While this is a significant improvement over the pure-random approach without any additional information, there is still a lot of room for improvement.

The modeled-random approach used the same model as our combinatorial testing approach, but instead of generating t-way abstract test cases using ACTS, it used MS Excel to generate the random abstract test cases.

The results in Figure 4.34 show that our approach achieved higher code coverage than the modeled-random approach, which further achieved higher code coverage than the pure-random approach. The results in Figure 4.34 also show that 3-way testing achieved higher code coverage than 2-way testing.

After we executed the test cases, we inspected the faults to see how many versions we have killed by looking at the source code. (We did not look at the source code during the modeling process.) Some of the faults are only triggered by invalid inputs and since we focused on interaction testing, we excluded those faults that can only be triggered by single invalid values.

Table 4.76 shows the fault detection results of the different approaches. These results are largely consistent with the code coverage results. That is, our approach detected more faults than modeled-random testing, which further detected more faults than pure-random testing.

Table 4.76 Fault detection results

| Subject Programs | Ant | | ACTS | | Space | | Make | | Grep | |
|---|---|---|---|---|---|---|---|---|---|---|
| | killed | not killed | killed | not killed | Killed | not killed | killed | not killed | killed | not killed |
| pure-random1 | 1 | 5 | 0 | 1 | 12 | 20 | 2 | 17 | - | - |
| pure-random2 | 1 | 5 | 0 | 1 | 15 | 17 | 2 | 17 | - | - |
| modeled-random1 | 3 | 3 | 1 | 0 | 23 | 9 | 6 | 13 | 4 | 1 |
| modeled-random2 | 4 | 2 | 1 | 0 | 26 | 6 | 6 | 13 | 4 | 1 |
| 2-way | 4 | 2 | 1 | 0 | 28 | 4 | 6 | 13 | 4 | 1 |
| 3-way | 5 | 1 | 1 | 0 | 30 | 2 | 9 | 10 | 4 | 1 |

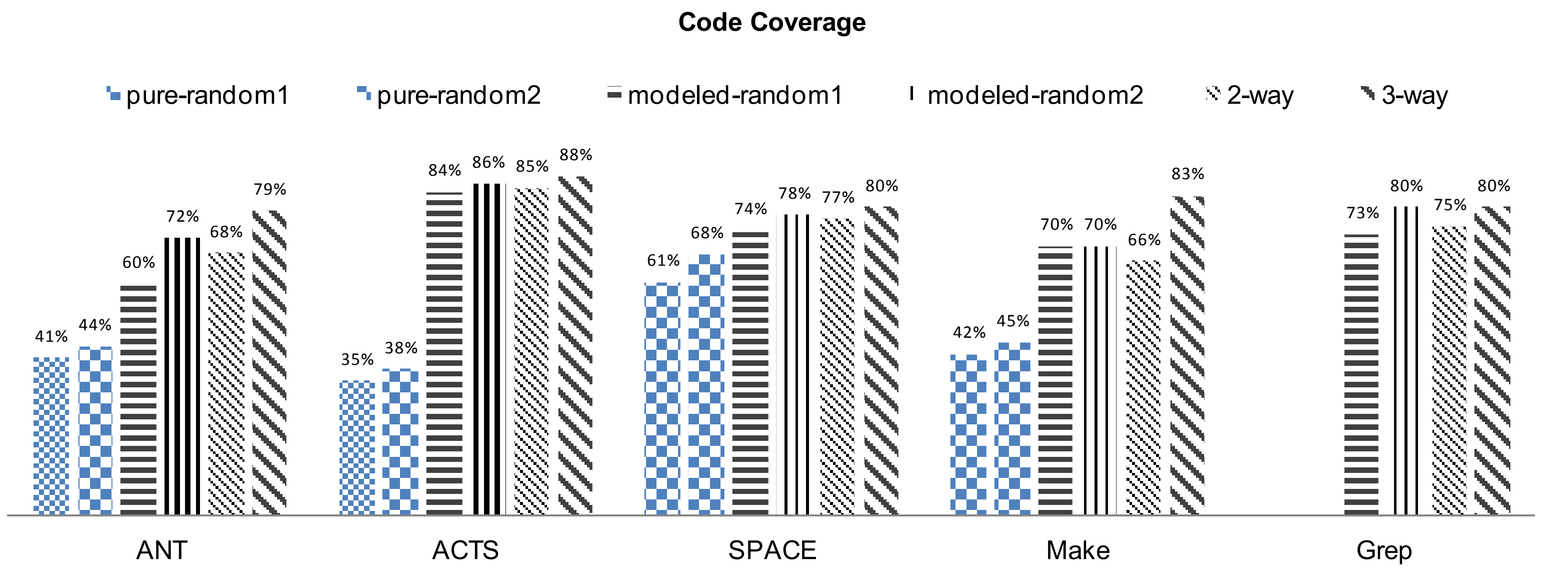


Figure 4.34 Code coverage results

The number of test cases for pure-random1 and modeled-random1 is the same as 2-way testing, and the number of test cases for pure-random2 and modeled-random2 is the same as 3-way testing.

The 3-way testing for *Space* was able to detect 93.7% of the faults. The faulty versions 12 and 18 (v12 and v18) were only killed by 3-way testing. None of our tests was able to detect v27. Version v33 was only detected by modeled-random testing (with the same number of test cases as 2-way testing).

To find out why some faults were not detected, we conducted an investigation to determine the strength of the faults mentioned above. Our investigation suggests that the strengths of fault for v12, v18, v27, and v33, are likely to be 4, 5, 7, and 5, respectively. This explains why they were not detected by some approaches. Note that a t-way test set also contains higher strength combinations. This is why v12 and v18 were detected by 3-way testing, even though they have a strength higher than 3. Similarly, the v33 was detected by modeled-random testing.

It is important to note that it can be difficult to determine the strength of fault for a large and/or complex program. In order to identify the suspicious parameters, we generated 20 more test cases by changing one parameter at a time and fixing the others. 8 out of 20 test cases were able to kill this version. By comparing the parameter values of these test cases, we were able to detect five suspicious parameters that could cause the fault.

To determine the strength of the fault, we generated 486 exhaustive test cases by fixing the value of the suspicious parameters. We randomly executed 10 out of 486 test cases and they all failed. Therefore, we believe that the strength of this fault is likely to be 5 [42]. We performed a similar investigation for other tests that were killed only by 3-way testing.

Although a 3-way test set guarantees to kill the faulty version when the fault strength does not go over 3, but it is possible that a 3-way test set kills a version with fault strength greater than 3.

The 3-way testing for *Make* was able to detect 48% of the faults. To find out why the faults were not detected, we increased the strength to 4. The 4-way testing was able to detect 73% of the faults. As the strength of t was increased our fault detection improved. We didn't try the 5-way testing due to the huge number of test cases, but we examined the fault that was missed by some of the 4-way test set. For three faults, we were not able to find any test case to expose them; As a result, we contact the SIR website administrator to see whether all the seeded faults are exposable. The administrator stated that there are some faults in the repository that are not detected by any tests and/or developing a test to expose them is time consuming. This suggested that the remaining faults are either not exposable or they likely have higher strength.

In addition, for the Make subject, the modeled-random1 test set achieved more code coverage than 2-way test set. Also for the Grep subject the modeled-random2 achieved almost the same code coverage as 3-way test set. We conducted an investigation to evaluate the obtained results. We generated 10 different set for modeled-random1 and modeled-random2. The results show that the modeled-random testing can achieve unpredictable code coverage as compared to the t-way testing with the same number of test cases. Therefore, the result of the modeled-random testing, unlike t-way testing, is not reliable and it can change from one test set to another.

The results of 10 different test sets suggest that the t-way testing have a better performance than the average of modeled-random. Also for each modeled-random set we calculate how many t-way combinations are covered. The results show that on average with the same number of test cases as t-way testing, modeled-random covers more than 90% of (t-1)-way combinations, 86% of t-way combinations, and 49% of (t+1)-way. This explains in part why the modeled-random approach achieved code coverage competitive to our combinatorial testing approach.

CHAPTER 5

# CONCLUSION AND FUTURE WORK

## 5.1 Conclusion

In this dissertation we presented an input space modeling methodology for combinatorial testing. Input space modeling is problem zero of combinatorial testing, and it determines to a large extent the effectiveness of combinatorial testing. The key idea of our methodology is to consider the modeling process as two steps, input structure modeling and input parameter modeling. We mainly consider the graph structure, which is further divided into graphs without loop and graphs with loop. We also suggested some guidelines to perform unit and integration testing based on the graph structure. We believe that input structure modeling is essential to manage complex input spaces such as those represented by XML files.

We also reported case studies of applying our methodology to five real-life programs: *ACTS*, *Apache Ant, Space, Make,* and *Grep*. We compared combinatorial testing based on the proposed methodology to two random approaches: pure-random and modeled-random. The modeled-random approach used the same model as our combinatorial testing approach, but instead of t-way it generated random abstract test cases.

We measured the effectiveness of these approaches in terms of code coverage and number of faults they can detect. The results of fault detection and code coverage were highly consistent. The results showed that our approach, which achieved 80.6% statement coverage and 77.7% fault detection, is more effective than modeled-random testing, which achieved 70% statement coverage and 61.9% fault detection, and both are significantly more effective than pure-random testing, which achieved 45% statement coverage and 30.5% fault detection on average. This not only suggests that input space modeling is an essential step in the combinatorial testing but also shows the effectiveness of combinatorial testing in practice.

In addition, we noticed that a t-way test set guarantees to kill faults with strength equal to t but it is possible to kill faults with greater strength because it contains higher strength combinations. Therefore, some faults with strength higher than our t-way test set will be detected.

The pure-random approach without any additional information only achieved 22% statement coverage on average. The pure-random approach with added information achieved 45% statement coverage on average. Despite the success of the second approach over the pure-random approach without any additional information, there still exist areas for further enhancements.

Moreover, t-way testing is more reliable and stable in terms of code coverage than modeled-random testing, which has already lost its benefit after we modeled the system.

## 5.2 Future work

There are three venues to continue our research work:

• Experiments on larger and/or more complex real-life programs. Although we have conducted experiments on several applications of significant size, it is still valuable to evaluate the effectiveness of our approach on even larger and/or more complex programs. For example, experiments conducted on web-services such as those developed using WS-BPEL will further evaluate the effectiveness of our approach.

• Develop a set of guidelines to help practitioners apply combinatorial testing in practice. The goal is to introduce a set of guidelines that can be used by practitioners to apply combinatorial testing in practice and to systematically identify parameters, values, constraints, and relations based on the program specification.

• Automatic identification of parameters, values, relations, and constraints. In the future, we will investigate on how to automatically identify parameters, values, relations, and constraints from other sources such as the design documents, xml schema files, or the actual source code. Specifications are often expressed in many different styles and formats. It is not practical to introduce rules that can be used as a mechanical approach to extract information from specifications. But formal documents follow certain rules. For example in the decision point of an activity diagram it is likely that the variable and its different choices can be identified as a parameter with its associated values respectively. In general, the automation tool can be helpful in giving suggestions for a given model and reducing the amount of manual modeling work that is required.

REFERENCES

[1]. Grindal, M. and Offutt, J., Input Parameter Modeling for Combination Strategies in Software Testing, Proceedings of the IASTED International Conference on Software Engineering (SE2007), Innsbruck, Austria, 13-15, pages 255-260, Feb 2007.

[2]. T. Chen, P.-L. Poon, S.-F. Tang, and T. Tse. On the Identification of Categories and Choices for Specification-based Test Case Generation. Information and Software Technology, 46(13):887–898, 2004.

[3]. M. Grochtmann and K. Grimm. Classification Trees for Partition Testing. Journal of Software Testing, Verification, and Reliability, 3(2):63–82, 1993.

[4]. T. J. Ostrand and M. J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. Communications of the ACM, 31(6):676–686, June 1988.

[5]. B. Beizer. Software Testing Techniques. Van Nostrand Reinhold, 1990.

[6]. Wenhua Wang, Sreedevi Sampath, Yu Lei, Raghu Kacker. An Interaction-Based Test Sequence Generation Approach for Testing Web Applications, IEEE International Conference on High Assurance Systems Engineerng, December 2008.

[7]. Clover: Code Coverage Tool for Java. http://www.cenqua.com/clover/.

[8]. Kuhn R, Wallace D, Gallo A. Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering; 30(6):418–421, 2004.

[9]. C. Nie and H. Leung. A survey of combinatorial testing. ACM Computing Surveys (CSUR), 43:11:1–11:29, 2011.

[10]. Krishnan, R.,Krishna, S. M., Nandhan, P. S.. Combinatorial Testing: Learnings From Our Experience. Sigsoft Softw. Engin. Notes 32, 3, 1–8., 2007.

[11]. Burr, K. and Young, W.. Combinatorial Test Techniques: Table -based Automation, Test Generation, And Code Coverage. In Proceedings Of The International Conference On Software Testing Analysis And Review. 503–513, 1998.

[12]. http://csrc.nist.gov/groups/SNS/acts/documents/comparison-report.html

[13]. Xu, L., Xu, B., Nie, C., Chen, H., and Yang, H. A Browser Compatibility Testing Method Based On Combinatorial Testing. In Proceedings of the International Conference on Web Engineering Icwe. Springer, Berlin, 310–313, 2003.

[14]. Williams, A. W. And Probert, R. L.. A Practical Strategy for Testing Pair-wise Coverage of Network Interfaces. In Proceedings of the 7th International Symposium on Software Reliability Engineering (Issre'96). Ieee Computer Society, Los Alamtos, Ca, 246, 1996.

[15]. Burroughs, K., Jain, A., and Erickson, R.. "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. 745–752., 1994.

[16]. A. M. Memon and Q. Xie, Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software, IEEE Transactions on Software Engineering, vol. 31, no. 10, pp. 884–896, 2005.

[17]. http://ant.apache.org/manual/

[18]. http://www.gnu.org/software/grep/manual/

[19]. http://sourceforge.net/projects/gcov-eclipse/

[20]. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel, Empirical Software Engineering: An International Journal, Volume 10, No. 4, pages 405-435, 2005.

[21]. Borazjany, Mehra N., Yu Lei.,et al. "Combinatorial Testing of ACTS: A Case Study." In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pp. 591-600. IEEE, 2012.

[22]. Richard Kuhn, Raghu Kacker, Yu Lei., Combinatorial and Random Testing Effectiveness for a Grid Computer Simulator presented at the Mod Sim World, Virginia, USA, 2009.

[23]. M. Brcic and D. Kalpic, Combinatorial testing in software projects. MIPRO, 2012 Proceedings of the 35th International Convention , vol., no., pp.1508-1513, 21-25 May 2012.

[24]. P. J. Schroeder, P. Bolaki, and V. Gopu, Comparing the fault detection effectiveness of n-way and random test suites, International Symposium on Empirical Software Engineering, 2004. ISESE '04. Proceedings, pp. 49– 59, 2004.

[25]. Jones, James A., Mary Jean Harrold, and John Stasko. "Visualization of test information to assist fault localization." In Proceedings of the 24th international conference on Software engineering, pp. 467-477. ACM, 2002.

[26]. http://oxygenxml.com/xml_developer.html

[27]. C. Lott, A. Jain, S. Dalal, Modeling Requirements for Combinatorial Software Testing, SIGSOFT Softw. Eng. Notes,30:1-7, 2005

[28]. Itai Segall, Rachel Tzoref-Brill, Aviad Zlotnick, Common Patterns in Combinatorial Models , 1st International Workshop on Combinatorial Testing (in conjunction with ICST'12), 2012.

[29]. Itai Segall, Rachel Tzoref-Brill, Aviad Zlotnick, Simplified Modeling of Combinatorial Test Spaces, 1st International Workshop on Combinatorial Testing (in conjunction with ICST'12), 2012.

[30]. Elke Salecker, Sabine Glesner, Combinatorial Interaction Testing for Test Selection in Grammar-Based Testing, IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012.

[31]. D. Richard Kuhn, Raghu N. Kacker, Yu Lei , Practical Combinatorial Testing, National Institute of Standards and Technology, 2010.

[32]. Bryce, R. C., Lei, Y., Kuhn, D. R. & Kacker, R. Combinatorial Testing, Handbook of Research on Software Engineering and Productivity Technologies: Implications of Globalization. IGI Global, 196-208., 2010

[33]. W. Wang, Y. Lei, D. Liu, D. Kung, C. Csallner, D. Zhang, R. Kacker and R. Kuhn, "A combinatorial approach to detecting buffer overflow vulnerabilities", Proceedings of 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), June 2011.

[34]. W. E. Wong, Y. Lei, Reachability Graph-Based Test Sequence Generation for Concurrent Programs, International Journal on Software Engineering and Knowledge Engineering, 18(6):803-822, Sept. 2008.

[35]. Y. Lei, R. Carver, R. Kacker, D. Kung, “A Combinatorial Strategy for Testing Concurrent Programs”, Journal of Software Testing,Verification, and Reliability, 17(4):207-225, 2007.

[36]. Qu, Xiao, Myra B. Cohen, and Katherine M. Woolf. "Combinatorial interaction regression testing: A study of test case generation and prioritization." Software Maintenance, 2007. ICSM 2007. IEEE International Conference on. IEEE, 2007.

[37]. S. Dalal and C. L. Mallows, "Factor-Covering Designs for Testing Software," Technometrics, vol. 50, no. 3, pp. 234-243, 1998.

[38]. W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of Test Set Size and Block Coverage on Fault Detection Effectiveness," in Proceedings of the Fifth IEEE International Symposium on Software Reliability Engineering. Monterey, CA, pp. 230-238., 1994.

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

[40]. D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The Combinatorial Design Approach to Automatic Test Generation," IEEE Software, vol. 13, no. 5, pp. 83-88, 1996.

[41]. Yin, Huifang, Zemen Lebne-Dengel, and Yashwant K. Malaiya. "Automatic test generation using checkpoint encoding and antirandom testing." In Proc. The Eighth International Symposium On Software Reliability Engineering, pp. 84-95. IEEE, 1997.

[42]. Zhang, Zhiqiang, and Jian Zhang. "Characterizing failure-causing parameter interactions by adaptive testing." In Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 331-341. ACM, 2011.

[43]. J. Czerwonka. “Pairwise Testing in Real World”. In Proc. 24th Pacific Northwest Software Quality Conference (PNSQC’06), pages 419–430, 2006.

[44]. M. Grindal, J. Offutt, and S.F. Andler, “Combination Testing Strategies: A Survey”, Software Testing, Verification, and Reliability, 15 (3): 167-199, 2005.

[45]. L. S. G. Ghandehari, Y. Lei, T. Xie, R. Kuhn, and R. Kacker., Identifying Failure-Inducing Combinations in a combinatorial test set. In Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12). IEEE Computer Society, Washington, DC, USA, 370-379, 2012.

[46]. J. Jones and M. Harrold, “Empirical evaluation of the tarantula automatic fault-localization technique”, In Proceeding IEEE/ACM Automated software engineering, N.Y., USA, 2005.

[47]. D. R. Kuhn and V. Okum., Pseudo-Exhaustive Testing for Software. InProceedings of the 30th Annual IEEE/NASA Software Engineering Workshop (SEW '06). IEEE Computer Society, Washington, DC, USA, 153-158, 2006.

[48]. D. R. Kuhn, D. Wallace, and A. Gallo, Software Fault Interactions and Implications for Software Testing, IEEE Transactions on Software Engineering, 30(6): 418-421, 2004.

[49]. B. Smith, M.S. Feather, and N. Muscettola. Challenges and Methods in Testing the Remote Agent Planner, Proc. Fifth Int’l Conf. Artificial Intelligence Planning Systems, 2000.

[50]. E. Wong and V. Debroy, “A survey on software fault localization,” Technical Report UTDCS-45-09, Department of Computer Science, University of Texas at Dallas, Nov. 2009.

[51]. E. Soechting, K. Dobolyi, and W. Weimer., Syntactic Regression Testing for Tree-Structured Output. In Proc. Int. Symp. on Web Systems Evolution (WSE'09). IEEE Computer Society, 2009.

[52]. Zhiqiang Lin , Xiangyu Zhang,, Input Syntactic Structure. Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, November 09-14, Atlanta, Georgia, 2008.

[53]. Pretschner, A.; Mouelhi, T.; Le Traon, Y., Model-Based Tests for Access Control Policies, International Conference on Software Testing, Verification, and Validation pp. 338-347, 2008.

[54]. Itai Segall, Rachel Tzoref-Brill, Interactive Refinement of Combinatorial Test Plans, International Conference on Software Engineering (ICSE'12), 2012.

[55]. J. Pan, The Dimensionality of Failures—A Fault Model for Characterizing Software Robustness, Proc. Int'l Symp. Fault-Tolerant Computing, June 1999.

[56]. D.R. Wallace and D.R. Kuhn, Failure Modes in Medical Device Software:An Analysis of 15 Years of Recall Data, Int'l J. Reliability, Quality and Safety Eng., vol. 8, no. 4, 2001.

[57]. Itai Segall, Rachel Tzoref-Brill, Aviad Zlotnick, Using binary decision diagrams for combinatorial test design, Proceedings of the 2011 International Symposium on Software Testing and Analysis.

[58]. Vilkomir, S.A.; Swain, W.T.; Poore, J.H.; Software Input Space Modeling with Constraints among Parameters, Computer Software and Applications Conference, 2009.

[59]. White, L. Almezen, H., Generating test cases for GUI responsibilities using complete interaction sequences, Software Reliability Engineering, 2000. Proceedings. 11th International Symposium, 110-121, ISSRE 2000.

[60]. Mandl, R. Corporation, C., Orthogonal Latin squares: an application of experiment design to compiler testing , Magazine Communications of the ACM CACM Homepage archive Volume 28 Issue 10, Oct. 1985.

[61]. Shams, M., Krishnamurthy, D., Far, B., A model-based approach for testing the performance of web applications , Proceedings of the Third International Workshop on Software Quality Assurance, 2006.

[62]. Burr, K. and Young, W. , Combinatorial Test TechniquesTable-based AutomationTest Generation And Code Coverage , In Proceedings Of The International Conference On Software Testing Analysis And Review. 503–513, October 1998.

[63]. Grindal, M., Offutt, J. and Mellin, J. , Managing Conflicts when Using Combination Strategies to Test Software., Proceedings of the 18th Australian Conference on Software Engineering (ASWEC2007), Melbourne, Australia, 10-13 April 2007.

[64]. Mats Grindal, Birgitta Lindström, Jeff Offutt, and Sten F. Andler., An Evaluation of Combination Testing Strategies, Empirical Software Engineering, 11(4):583-611, December 2006.

[65]. Mats Grindal, Evaluation of Combination Strategies for Practical Testing, PhD thesis, Skovde University, Sweden, 2007.

[66]. D.R. Kuhn and M.J. Reilly, An Investigation of the Applicability of Design of Experiments to Software Testing , Proc. 27th NASA/IEEE Software Eng.Workshop, Dec. 2002.

[67]. Dean Hoskins , Renée C. Turban , Charles J. Colbourn , Experimental designs in software engineering: d-optimal designs and covering arrays , Proceedings of the 2004 ACM workshop on Interdisciplinary software engineering research, November 05-05, 2004.

[68]. Shi, L., Nie, C., Xu, B. , A Software Debugging Method Based on Pairwise Testing , In Proceedings of the International Conference on Computational Science., 1088—1091, 2005.

[69]. Renée C. Bryce, Yu Lei, D. Richard Kuhn, Raghu Kacker ,Chapter 14 - Combinatorial Testing. En Software Engineering and Productivity Technologies, M. Ramachandran and R.A.d. Carvalho (eds.). Idea Group Publishing. p. 196-208, 2010.

[70]. Renée C. Bryce , Atif M. Memon,Test Suite Prioritization by Interaction Coverage , Workshop on Domain specific approaches to software test automation: in conjunction with the 6th ESEC/FSE joint meeting, p.1-7, September 04-04, 2007.

[71]. Renée C. Bryce , Charles J. Colbourn,,Test prioritization for pairwise interaction coverage , Proceedings of the 1st international workshop on Advances in model-based testing, p.1-7, May 15-21, 2005.

[72]. I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, A. Iannino, Applying design of experiments to software testing , Proceeding ICSE '97 Proceedings of the 19th international conference on Software engineering, 1997.

[73]. Pak-Lok Poon, Sau-Fun Tang, T.H. Tse, and T.Y. Chen, CHOCLAT: a CHOiCe reLATion framEwork for specification-based testing, Communications of the ACM 53 (4): 113-118, 2010.

[74]. Chen, T. Y., Poon, P.-L., Tang, S.-F., Tse, T. H., a DividE-and-conquer methodology for identifying categorieS, choiceS, and choicE Relations for Test case generation , IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2011.69, 2011.

[75]. Gary E., B.Math., Requirements Based Testing Cause-Effect Graphing , Software Testing Services, 2010.

[76]. Matt Archer , Test Case Design with Classification Trees (Sample Book Chapter), A WebPage posted Posted by Matt Archer on March 7, 2012.

[77]. Pak-Lok Poon, Tsong Yueh Chen, T.H. Tse, Comparing between CHOC'LATE and classification tree , Reliable Software Technologies: Ada-Europe 2012.

[78]. Glenford J. Myers., The Art of Software Testing, John Wiley & Sons, NY, 1979.

[79]. Chen, T.Y.; Poon, P.L., Classification-Hierarchy Table: A Methodology for Constructing the Classification Tree, Software Engineering Conference, Pages 93-104, Australian, 1996.

[80]. Balakrishnan Ramadoss, Paramasivam Prema, An Approach for Merging Two Classification-Trees, Proceedings of the IEEE International Advance Computing Conference, 1829-1834, 2009.

[81]. Boris Beizer, Software Testing Techniques. Van Nostrand Reinhold, NY, 2nd edition,. ISBN 0-442-20672-0, 1990.

[82]. M. Balcer, W. Hasling, T. Ostrand, Automatic generation of test scripts from formal test specifications, SIGSOFT Software Engineering Notes Volume 14 Issue 8, December 1989.

[83]. Paul Ammann and Jeff Offutt,Using Formal Methods To Derive Test Frames in Category-Partition Testing, Ninth Annual Conference on Computer Assurance (COMPASS 94), IEEE Computer Society Press pages 69-80, Gaithersburg, Maryland, June 1994.

[84]. Gutiérrez, J. J., Escalona M. J., Mejías M., Torres, J., Generation of test cases from functional requirements. A survey, 4th Workshop on System Testing and Validation. Germany, 2006.

[85]. Praveen Ranjan Srivastava, Parshad Patel, Siddharth Chatrola, Cause effect graph to decision table generation, SIGSOFT Software Engineering Notes, Volume 34, Number 2, March 2009.

[86]. Man F. Lau, Yuen T. Yu, An extended fault class hierarchy for specification-based testing, ACM Transactions on Software Engineering and Methodology (TOSEM), v.14 n.3, p.247-276, July 2005.

[87]. Amit Paradkar, K.C. Tai, M.A. Vouk, Specification-based testing using cause-effect graphs, Annals of Software Engineering 4. 133–157, 1997.

[88]. Phil Stocks, David Carrington,A Framework for Specification-Based Testing, IEEE Transactions On Software Engineering, Vol. 22, No. 11, November 1996.

[89]. Tang, Sau F., Phd Thesis: Identifying categories and choices for software testing based on informal specifications, A thesis submitted in total fulfillment of the

requirements of the degree of Doctor of Philosophy, Swinburne University of Technology, 2009.

[90]. T.Y. Chen, Pak-Lok Poon, T.H. Tse,A Choice Relation Framework for Supporting Category-Partition Test Case Generation, IEEE Transactions On Software Engineering, Vol. 29, No. 7, July 2003.

[91]. T. J. Ostrand and M. J. Balcer.,, The Category-Partition Method for Specifying and Generating Functional Tests., Communications of the ACM, 31(6):676–686, June 1988.

[92]. Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence, “IPOG/IPO-D: Efficient Test Generation for Multi-way Combinatorial Testing”, Software Testing, Verification & Reliability, 18(3): 125-148, 2007.

[93]. Kruse, P.M.; Wegener, J. , Test Sequence Generation from Classification Trees., Information Systems and Technologies (CISTI), 2011 6th Iberian Conference, p. 1-4, 2011.

[94]. Cain, A; Chen, TY; Grant, D; Poon, PL; Tang, S-F; Tse, TH ,An automatic test data generation system based on the integrated classification-tree methodology , Software Engineering Research and Applications, v. 3026, p. 225-238, 2004.

[95]. Lu Luo , Software Testing Techniques, Institute for software research international Carnegie mellon university Pittsburgh, PA 15232 USA. 1-19, 2002.

[96]. Grindal, M., Handling Combinatorial Explosion in Software Testing. , Thesis Dissertation no 1073, Department of Computer and Information Science Linköpings universitet, 2006

[97]. T. Chen, P.-L. Poon, S.-F. Tang, and T. Tse, On the Identification of Categories and Choices for Specification-based Test Case Generation., Information and Software Technology, 46(13):887–898, 2004.

[98]. T. Chen, S.-F. Tang, P.-L. Poon, and T. Tse., Identification of Categories and Choices in Activity Diagrams., In Proceedings of the Fifth International Conference on Quality Software (QSIC 2005) 19-20 September 2005, Melbourne, Australia, pages 55–63. IEEE Computer Society, September 2005.

[99]. Lämmel, Ralf, and Wolfram Schulte. "Controllable combinatorial coverage in grammar-based testing." In *Testing of Communicating Systems*, pp. 19-38. Springer Berlin Heidelberg, 2006.

[100]. Y. K. Malaiya. , Getting the most out of black-box testing. , In Proceedings of the International Symposium on Software Reliability Engineering, (ISSRE’95), Toulouse, France, Oct, 1995, pages 86–95, Oct. 1995.

[101]. H. Yin, Z. Lebne-Dengel, and Y. K. Malaiya. , Automatic Test Generation using Checkpoint Encoding and Antirandom Testing , Technical Report CS-97- 116, Colorado State University, 1997.

[102]. Benattou,M., Bruel, J., Hameurlain, N.,, Generating Test Data from OCL Specification, in Proceedings of the ECOOP'2002 Workshop on Integration and Transformation of UML models (WITUML02), 2002.

[103]. A. A. Omar, F. A. Mohammed, survey of software functional testing methods, SIGSOFT Software Engineering Notes , Volume 16 Issue 2, April 1991.

[104]. Myers, G., Software Reliability: Principles and Practices, John Wiley, NY 1976.

[105]. Goodenough, J.B., Gerhart, S.L., Toward a theory of test data selection, IEEE trans. softw. Eng. SE-2, 156-173, June 1975.

[106]. Debra J. Richardson, Lori A. Clarke, A partition analysis method to increase program reliability, Proceedings of the 5th international conference on Software engineering, p.244-253, March 09-12, 1981.

[107]. Elaine J. Weyuker And Thomas J. Ostrand,Theories of Program Testing and the Application of Revealing Subdomains , IEEE Transactions On Software Engineering, Vol. Se-6, No. 3, May 1980.

[108]. Mehra N.Borazjany, Laleh Gh., Yu Lei, Raghu Kacker, and Rick Kuhn. "An Input Space Modeling Methodology for Combinatorial Testing", 2nd Intl. Workshop on Combinatorial Testing, Luxembourg, IEEE, Mar. 2013. (To be appeared on IWCT 2013)

[109]. Laleh Gh., Mehra N. Borazjany, Yu Lei, R.N. Kacker, and D.R. Kuhn, "Applying Combinatorial Testing to the Siemens Suite", 2nd Intl. Workshop on Combinatorial Testing, Luxembourg, IEEE, Mar. 2013. (To be appeared on IWCT 2013)

## BIOGRAPHICAL INFORMATION

Mehra Nouroz Borazjany received his B.S. Degree in Computer Engineering from Azad University of Tehran, Iran, in May 2003, M.S. Degree in Computer Architecture Engineering from Tehran University of Science and Technology, Iran, in May 2007, and PhD degree in Computer Science and Engineering from the University of Texas at Arlington in May 2013. Her research interests include input space modeling for combinatorial testing and automated software testing.