Statistical Methods for Test and Evaluation, Volume 3: Combinatorial Design using R

Statistical Methods for Test and Evaluation, Volume 3: Combinatorial Design using R

By Jeffrey Strickland, Ph.D.

ISBN: 978-1-304-42779-3

TABLE OF CONTENTS

Preface

There are three fundamental test designs for testing systems in a defense acquisition lifecycle. However, prior to 2009, there was a lack of rigor underlying test designs. To correct this situation, Scientific Test & Analysis Techniques (STAT) Center of Excellence (COE) was established. The goal of the STAT COE is to quantify and characterize system performance and provide information that reduces risk. Moreover, STAT COE assists in developing rigorous, defensible test strategies.

STAT relied heavily on Design of Experiments (DOE) to inject rigor into test and evaluation (T&E). The goal was to more effectively characterize a System Under Test (SUT). While DOE achieves the this goal, it is not the only objective of T&E.

One of the key objections to testing as a form of software verification is that it is never possible to show that the SUT works for all possible inputs. The classical approach has been to apply design of experiments (DOE) techniques to Department of Defense (DoD) acquisition programs in Test and Evaluation (T&E). However, for deterministic systems, DOE may not be applicable because of the system's inability to produce any variation in the output with the same set of inputs. In such cases, we examine the use of combinatorial designs.

Combinatorial Test Design (CTD) is an optimal approach that helps us identify the tests that are likely to expose defects. CTD takes a systematic approach to modeling the things that need to be tested, then uses advanced mathematics to dramatically reduce the number of test cases while ensuring coverage of conditions and interactions. Used with Covering Arrays (CAs), CTD offer ways to build on existing techniques for input space partitioning, to provide more rigorous testing.

The advent of computers and telecommunication systems underlined the importance of thoroughly testing software and hardware-software systems. Software engineers tried using orthogonal arrays (OAs) to include all pairs of test settings, but quickly realized the limitations of OA-based test suites. Often, an OA matching the required combinatorial test structure did not exist, and OA-based test suites tended to include invalid (nonexecutable) test cases.

System failures often result from the interaction of conditions that might be innocuous individually, so combinatorial testing can be effective for domains with many interacting parameters, such as aerospace applications. Research suggests that in some circumstances, covering four-way to six-way combinations can be nearly as effective as testing all possible combinations, often dramatically reducing the number of test cases.

Because of these results, interest is growing in combinatorial testing, but many testers do not fully understand it. The method has its roots in DOE, a methodology for conducting controlled experiments in which a system is exercised for chosen test settings of various input variables, or factors. The corresponding values of one or more output variables, or responses, are measured to generate information for improving the performance of a class of similar systems.

One of the two purposes for randomized block designs, is to group heterogeneous experimental units together in homogeneous subgroups called blocks. This increases the power or precision for detecting differences in treatment groups. The overall F-test for comparing treatment means, in a randomized block design, is a ratio of the variability among treatment means to the variability of experimental units within the homogeneous blocks.

When combined with Covering Arrays, CTD offers ways to build on existing techniques for input space partitioning, to provide more rigorous software testing.

Other Technical Books by the Author

32. *Fundamentals of Combat Modeling*, Copyright © 2011 by Jeffrey S. Strickland. Lulu.com, ISBN 978-1-257-00583-3.

Non-Technical Books by the Author

33. *Knights of the Cross*, Copyright © 2011 by Jeffrey S. Strickland. Lulu.com, ISBN 978-1-105-35162-4.
34. A Mason's Mason, by Jeffrey S. Strickland. Lulu.com 2024 by Jeffrey S. Strickland. Lulu, Inc., ISBN 978-1-312-03214-9
35. *Weird Scientists – the Creators of Quantum Physics*, Copyright © 2011 by Jeffrey S. Strickland. Lulu.com, ISBN 978-1-257-97624-9
36. *Albert Einstein: "Nobody expected me to lay golden eggs"*, Copyright © 2011 by Jeffrey S. Strickland. ISBN 978-1-257-86014-2.
37. *The Men of Manhattan: Creators of the Nuclear Era*, Copyright © 2011 by Jeffrey S. Strickland. Lulu.com, ISBN 978-1-257-76188-3.
38. *The Devil Did Not Make Me Do It*, Copyright © 2018 by Jeffrey S. Strickland, Printed by Lulu, Inc., ISBN 9781365982149.
39. *To Watch Over Them Day and Night*, Copyright © by Jeffrey Strickland, Printed by Lulu, Inc. ISBN 9781365880759.
40. *Quantum Phaith*, Copyright © 2011 by Jeffrey Strickland, Printed by Lulu, Inc., ISBN 9781257645619
41. *Quantum Hope*, Copyright © 2015 by Jeffrey Strickland, Printed by Lulu, Inc., ISBN 9781329160781.
42. *Quantum Love*, Copyright © 2017 by Jeffrey Strickland, Printed by Lulu, Inc., ISBN 9781365835551
43. *I Rode with Wallace*, Copyright © 2015 by Jeffrey Strickland, Printed by Lulu, Inc., ISBN 9781329565661.
44. *Handbook of Handguns*, Copyright © 2013 by Jeffrey Strickland, Printed by Lulu, Inc., ISBN 9781300973294.
45. *LinkedIn Memoirs*, Copyright © 2014 by Jeffrey Strickland, Printed by Lulu, Inc., ISBN 9781312717329.
46. *Dear Mr. President*, Copyright © 2012 by Jeffrey Strickland, Printed by Lulu, Inc., ISBN 9781105555145.

1 Introduction to *Software Test Design*

A software tester can perform various testing methods to make sure that a software application is developed to fulfill **minimal viable Products** (MVPs) or the requirements set forth by a **Software Capabilities Development Document** (SW-CDD). However, if we have a highly complex software application with a lot of input possibilities, we may have a lot of testing to do. For example, search functions have multiple options that may have complex interactions. Exhaustive testing is impossible because many billions of potential tests would have to be performed. It would take a long time to design and execute all the test scenarios.

Structural coverage adds some rigor to the process by establishing formally defined criteria for some notion of test completeness, but even full coverage, however defined, may miss faults related to rare inputs that were not included in the test suite. NIST performed a longevity study that discovered the number of factors involved when software failures are small, and they saw no failures involving more than 6 variables. Surprisingly, prior to their study, no one had previously looked at greater than 2-way interactions. (Kuhn D. R., 2023)

One of the key objections to testing as a form of software verification is that it is never possible to show that the **system under test** (SUT) works for all possible inputs. It is also difficult to provide meaningful statements about the adequacy of a test set for verifying that the SUT works correctly.

Conventional structural coverage measures, typically statement or branch coverage, leave much to be desired. Even if all statements are executed and all branches are taken, there is no guarantee that the input space has been covered adequately for fault detection. A latent error may show up later with the appearance of a very rare combination of conditions that was not included in testing.

Methods of systematically partitioning the input space have been studied extensively, but most necessarily involve a good deal of subjective judgement, and do not provide quantitative measures of completeness. **Combinatorial methods** and **Covering Arrays** offer ways to build on existing techniques for input space partitioning, to provide more rigorous testing.

Combinatorial Test Design (CTD) is an optimal approach that helps us identify the tests that are likely to expose defects. CTD takes a systematic approach to modeling the things that need to be tested, then uses advanced mathematics to dramatically reduce the number of test cases while ensuring coverage of conditions and interactions.

1.1 Understanding Combinatorial Testing

System failures often result from the interaction of conditions that might be innocuous individually, so combinatorial testing can be effective for domains with many interacting parameters, such as aerospace applications. Research suggests that in some circumstances, covering four-way to six-way combinations can be nearly as effective as testing all possible combinations, often dramatically reducing the number of test cases. (Kuhn, Wallace, & Gallo, 2004)

Because of these results, interest is growing in combinatorial testing, but many testers do not fully understand it. The method has its roots in design of experiments (DOE), a methodology for conducting controlled experiments in which a system is exercised for chosen test settings of various input variables, or factors. The corresponding values of one or more output variables, or responses, are measured to generate information for improving the performance of a class of similar systems. These methods were developed in the 1920s and 1930s to improve agricultural production, and later adapted for other industries.

Early testing with DOE revealed its weaknesses in testing software applications, which led to the use of covering arrays (CAs)—these are still the primary vehicle for implementing combinatorial testing in software systems.

1.1.1 Design of Experiments (DOE) Method

DOE’s objective is to improve the mean response over replications. Japanese quality engineering guru, Genichi Taguchi, (Taguchi, 1987) promulgated a variation of these methods using **orthogonal arrays** (OAs), in which every factor-level combination occurs the same number of times. He used OAs in industrial experiments, aiming to determine the test settings at which the variation from uncontrolled factors was the lowest. (Kacker, 1985) (Phadke, 1989)

The advent of computers and telecommunication systems underlined the importance of thoroughly testing software and hardware-software systems. Software engineers tried using OAs to include all pairs of test settings, but quickly realized the limitations of OA-based test suites. Often, an OA matching the required combinatorial test structure did not exist, and OA-based test suites tended to include invalid (nonexecutable) test cases.

1.1.2 Covering Arrays (CAs)

OA-based testing limitations led to the use of CAs, which can be constructed for any strength testing—unlike OAs, which are generally limited to strengths 2 and 3. In general, OA use is now relegated to statistical questions, while software testing relies more on covering arrays. CAs have several advantages over OAs:

- They can be constructed for any combinatorial test structure of unequal test setting numbers.
- If an OA exists for a combinatorial test structure, a CA of the same number or fewer test cases is also possible.
- Test suites can exclude invalid combinations.

Definition: A Covering Array (CA) is basically a matrix that includes all t- way combinations of values for some specified interaction level t. A fixed- value CA, $\mathrm{CA}(N;\ t, k, v)$, is an $N \times k$ matrix of elements from a set of v symbols $\{0, 1, \dots, (v - 1)\}$ such that every set of t columns contains each possible t-tuple of elements at least once. The positive integer t is the strength of the covering array.

A mixed-value, or mixed-level, covering array is a generalization of the fixed- value CA that allows columns to have different numbers of distinct elements. Most software testing problems require mixed-level arrays because parameters might have different numbers of values or equivalence classes.

To illustrate a CA, consider testing a four-parameter function that inserts text into a document. The four parameters and representative values are location:

`{start, middle, end}, text_size: {small, medium, max}, document_size: {small, medium, max}, and selection_method: {mouse, edit_menu, keyboard}.`

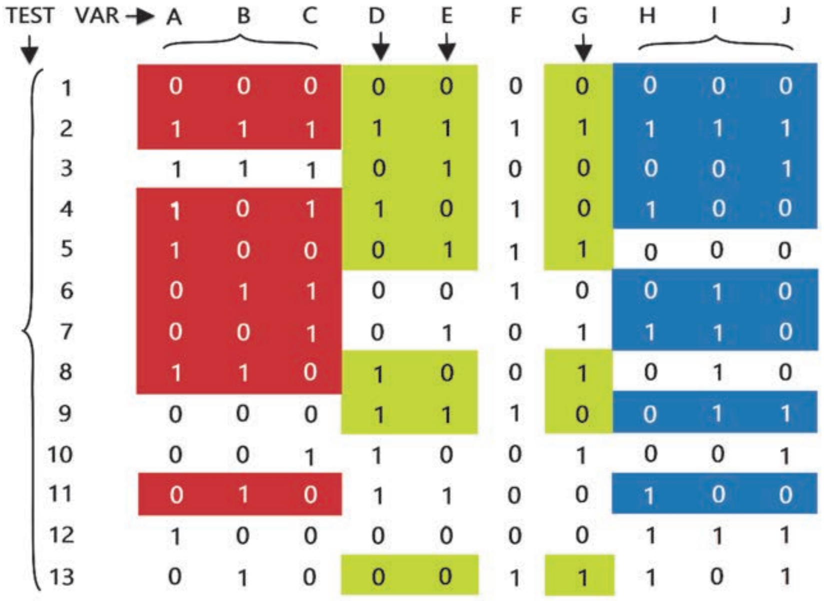
Thus, 3^4 (81) possible combinations will exhaustively test these values. However, the nine tests in ***Table 1-1*** can test all two-way interactions—that is, all possible value pairs occur at least once among the tests.

Table 1-1. Nine tests that cover all two- way interactions (pairwise).

Test	Location	Text_size	Doc_size	Select_method
1	Start	Small	Small	Mouse
2	Start	Medium	Medium	Edit_menu
3	Start	Max	Max	Keyboard
4	Middle	Small	Max	Edit_menu
5	Middle	Medium	Small	Keyboard
6	Middle	Max	Medium	Mouse
7	End	Small	Medium	Keyboard
8	End	Medium	Max	Mouse
9	End	Max	Small	Edit_menu

Reducing the test set from 81 to 9 is not that impressive, but a larger example illustrates a more dramatic decrease. Suppose the test is of a manufacturing automation system with 20 controls, each with 10 possible settings—a total of 10^{20} combinations. With a covering array, only 162 tests can cover all pairs of these values.

Figure 1-1 gives another example, this one of a test with a three-way covering array for 10 variables with two values each. As ***Figure 1-1*** shows, any three array columns contain all eight possible values for three binary variables. For example, taking columns A, B, and C, all eight possible three-way combinations (000, 001, 010, 011, 100, 101, 110, 111) occur somewhere in the three columns together. Any combination of three columns chosen in any order will also contain all eight possible values. Collectively, therefore, this test set will exercise all three- way combinations of input values in only 13 tests, as compared with 1,024 for exhaustive coverage.



TEST ↓ VAR →	A	B	C	D	E	F	G	H	I	J
1	0	0	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	1	1	1	1
3	1	1	1	0	1	0	0	0	0	1
4	1	0	1	1	0	1	0	1	0	0
5	1	0	0	0	1	1	1	0	0	0
6	0	1	1	0	0	1	0	0	1	0
7	0	0	1	0	1	0	1	1	1	0
8	1	1	0	1	0	0	1	0	1	0
9	0	0	0	1	1	1	0	0	1	1
10	0	0	1	1	0	0	1	0	0	1
11	0	1	0	1	1	0	0	1	0	0
12	1	0	0	0	0	0	0	1	1	1
13	0	1	0	0	0	1	1	1	0	1

Figure 1-1. Testing with a three- way covering array. The tests must cover 10 binary variables with two values each, which would take 210, or 1,024, tests to cover exhaustively. However, testing with the three- way covering array takes only 13 tests (rows). Highlighted numbers correspond to the eight possible three-way combinations in each set of three columns.

Testers can generate similar arrays to cover combinations of more parameters. In general, the number of t-way combinatorial tests required is proportional to $v^t \log n$, for n parameters with v possible values each. It is best to limit the number of equivalence classes or discrete parameter values to about 10 or fewer, but applications with a larger number of parameters do not present a problem. In general, combinatorial testing will cover applications with up to a few hundred parameters, which is sufficient for most software testing problems.

Methods for constructing CAs fall into three categories: **algebraic**; (Sloane, 1993) **metaheuristic**, such as simulated annealing and tabu search; (Cohen, Colbourn, & Ling, 2003) (Colbourn, 2004) and greedy search. (Lei & Tai, 1998) (Bryce, Colbourn, & Cohen, 2005) For certain special combinatorial test structures, algebraic methods are extremely fast and can produce very compact CAs, but many testing problems do not fit the requirements of algorithms in this category.

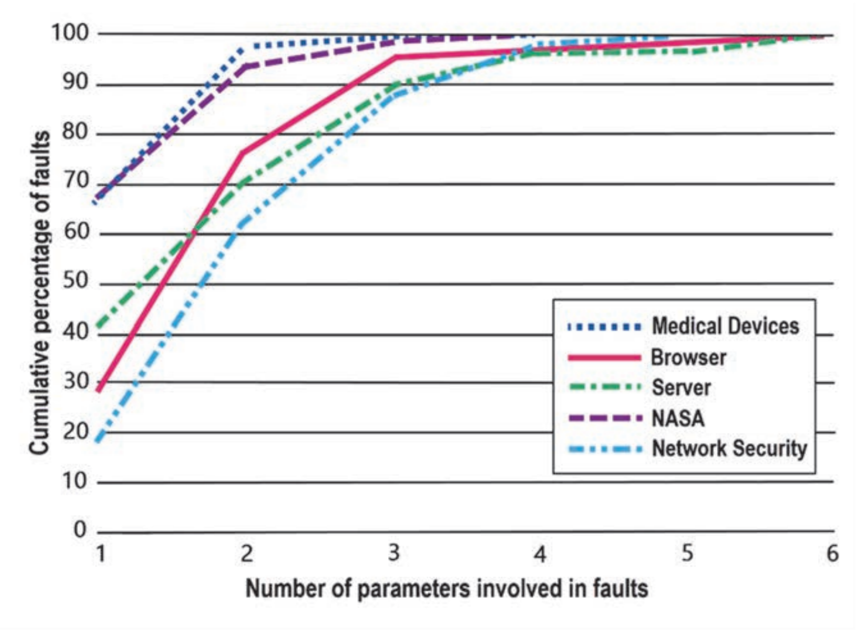


Figure 1-2. Percentage of errors triggered by t-way interactions. Most faults tend to be triggered by a single value, with the fewest being triggered by six-way interaction.

Metaheuristic methods have produced some of the smallest CAs, but they can be slow, particularly for large testing problems. Greedy methods are faster than metaheuristic methods and can handle arbitrary test structures, but they might not always produce the smallest CAs. However, greedy methods are also convenient for excluding or including certain combinations because of constraints among parameter values. Typically, they are the most widely used category in tools to support practical problems.

1.1.3 Empirical Basis

Most failures result from a single parameter or a two-way interaction between parameters, but what about the remaining faults? In a study of actual software faults in a variety of domains, (Kuhn, Wallace, & Gallo, Software Fault Interactions and Implications for Software Testing, 2004) faults followed the Interaction Rule, which says that most faults are triggered by a single value or an interaction between two parameter values, and progressively fewer are triggered by three- , four- , five- , and six-way interactions. (Kuhn, Kacker, & Lei, 2013)

Figure 1-2 summarizes the results of this study. In the Web server application (Server), a single value caused roughly 40 percent of the failures, such as a file name exceeding a certain length. Another 30 percent were triggered by the interaction of two parameters. Thus, three or fewer parameters triggered a total of almost 90 percent of the failures in that application. Curves for the other applications have a similar shape, reaching 100 percent detection between four- and six-way interactions.

These results suggest that combinatorial testing, which exercises strong interaction combinations, can be an effective approach to high integrity software assurance. The key insight in combinatorial testing is this: (Kuhn, Wallace, & Gallo, Software Fault Interactions and Implications for Software Testing, 2004)

1.2 Types of Test Designs

Test design is often confused with DOE. DOE is a test design methodology and should not be confused with test design. Test designs are based on the objective of T&E. Generally, there are three primary test designs:

1. System Characterization
2. System Comparison
3. System Error Identification

Most often applied to hardware systems, characterization depends heavily on DOEs like full factorial with k factors and fractional factorial (see Montgomery (Design and Analysis of Experiments, 2019)).

System comparison requires the use of statistical parameter tests, like the nonparametric sign test and parametric t-test, with some underlying knowledge of probability distributions.

Most often applied to software systems, error identification relies on CTDs and CAs, which are our subjects.

1.3 CTD for Identifying Software Errors

Software testing is the primary application of CTD methods, including functionality testing and security vulnerabilities. Approximately 2/3 of vulnerabilities stem from implementation errors. So, what causes software failures?

- logic errors
- calculation errors
- Inadequate input checking
- interaction faults
- Etc.

CTD or t-way testing is a proven method for better testing at lower cost. Line Graph showing Cumulative percent of software failures. As mentioned in Sectio 1.1, the key insight underlying its effectiveness resulted from a series of studies by NIST from 1999 to 2004. NIST research showed that most software bugs and failures are caused by one or two parameters, with progressively fewer by three or more, which means that combinatorial testing can provide more efficient fault detection than conventional methods. Multiple studies have shown fault detection equal to exhaustive testing with a 20× to 700× reduction in test set size. New algorithms compressing combinations into a small number of tests have made this method practical for industrial use, providing better testing at lower cost. (Smith, et al., 2019a) (Smith, et al., 2019b) (Kuhn, Kacker, Feldman, & Witte, 2016)

1.4 Definitions

Combinatorial Design: A combinatorial design is a way of arranging elements from a set into specific structures (often called blocks or groups) according to certain rules. These designs are studied within combinatorics, a branch of mathematics that deals with counting, arrangement, and combination.

Block: In combinatorial designs, a block refers to a specific subset of elements chosen from a larger set, where each subset (block) is structured according to prescribed rules to study the arrangements and combinations fulfilling certain criteria.

A **balanced incomplete block design (BIBD)** is structured such that each block contains a specific number of elements, each element appears in a fixed number of blocks, and every pair of distinct elements appears together in exactly the same number of blocks.

Latin squares are a fundamental example of combinatorial designs, where each row and column contains a set of symbols exactly once, thereby exemplifying a balanced arrangement that is pivotal in the study and construction of more complex combinatorial structures.

Experiment (also called a Run) is an action where the experimenter changes at least one of the variables being studied and then observes the effect of his or her actions(s). Note the passive collection of observational data is not experimentation.

Experimental Unit is the item under study upon which something is changed. This could be raw materials, human subjects, or just a point in time.

Sub-Sample, Sub-Unit, or **Observational Unit** When the experimental unit is split, after the action has been taken upon it, this is called a sub-sample or sub-unit. Sometimes it is only possible to measure a characteristic separately for each sub-unit; for that reason they are often called observational units. Measurements on sub-samples, or sub-units of the same experimental unit, are usually correlated and should be averaged before analysis of data rather than being treated as independent outcomes. When sub-units can be considered independent and there is interest in determining the variance in sub-sample measurements, while not confusing the 8-tests on the treatment factors, the mixed model described in Section 5.8 should be used instead of simply averaging the sub-samples.

Independent Variable (Factor or Treatment Factor) is one of the variables under study that is being controlled at or near some target value, or level, during any given experiment. The level is being changed in some systematic way from run to run in order to determine what effect it has on the response(s).

Background Variable (also called a Lurking Variable) is a variable that the experimenter is unaware of or cannot control, and which could have an effect on the outcome of the experiment. In a well-planned experimental design, the effect of these lurking variables should balance out so as to not alter the conclusion of a study.

Dependent Variable (or the Response denoted by Y) is the characteristic of the experimental unit that is measured after each experiment or run. The magnitude of the response depends upon the settings of the independent variables or factors and lurking variables.

Effect is the change in the response that is caused by a change in a factor or independent variable. After the runs in an experimental design are conducted, the effect can be estimated by calculating it from the observed response data. This estimate is called the calculated effect. Before the experiments are ever conducted, the researcher may know how large the effect should be to have practical importance. This is called a practical effect or the size of a practical effect.

Replicate runs are two or more experiments conducted with the same set- tings of the factors or independent variables but using different experimental units. The measured dependent variable may differ among replicate runs due to changes in lurking variables and inherent differences in experimental units.

Duplicates refer to duplicate measurements of the same experimental unit from one run or experiment. The measured dependent variable may vary among duplicates due to measurement error, but in the analysis of data these duplicate measurements should be averaged and not treated as separate responses.

Experimental Design is a collection of experiments or runs that is planned in advance of the actual execution. The particular runs selected in an experimental design will depend upon the purpose of the design.

Confounded Factors arise when each change an experimenter makes for one factor, between runs, is coupled with an identical change to another factor. In this situation it is impossible to determine which factor causes any observed changes in the response or dependent variable.

Biased Factor results when an experimenter makes changes to an independent variable at the precise time when changes in background or lurking variables occur. When a factor is biased it is impossible to determine if the resulting changes to the response were caused by changes in the factor or by changes in other background or lurking variables.

Experimental Error is the difference between the observed response for a particular experiment and the long run average of all experiments con- ducted at the same settings of the independent variables or factors. The fact that it is called \error" should not lead one to assume that it is a mistake or blunder. Experimental errors are not all equal to zero because background or lurking variables cause them to change from run to run. Experimental errors can be broadly classified into two types: bias error and random error. Bias error tends to remain constant or change in a consistent pattern over the runs in an experimental design, while random error changes from one experiment to another in an unpredictable manner and average to be zero. The variance of random experimental errors can be obtained by including replicate runs in an experimental design.

1.5 Background – Fault Distribution.

Empirical data show that most failures are triggered by a single parameter value, or interactions between a small number of parameters, generally two to six, a relationship known as the interaction rule (see ***Figure 1-3***). An example of a single-value fault might be a buffer overflow that occurs when the length of an input string exceeds a particular limit. Only a single condition must be true to trigger the fault: input length > buffer size. A 2-way fault is more complex, because two particular input values are needed to trigger the fault.

One example is a search/replace function that only fails if both the search string and the replacement string are single characters. If one of the strings is longer than one character, the code does not fail, thus we refer to this as a 2-way fault. More generally, a t-way fault involves t such conditions.

Figure 1-4 shows the cumulative percentage of faults (y-axis) at $t = 1$ to 6 (x-axis) for various real applications (Kuhn, Dominguez, Kacker, & Lei, 2013)

We refer to the distribution of t-way faults as the fault profile. ***Figure 1-4*** shows the fault profile for a variety of fielded products in different application domains, and results for initial testing of a NASA database system. As shown in Figure 1, the fault detection rate increases rapidly with interaction strength, up to t=4. With the medical device applications, for example, 66% of the failures were triggered by only a single parameter value, 97% by single values or 2-way combinations, and 99% by single values, 2-way, or 3-way combinations. The detection rate curves for the other applications studied are similar, reaching 100% detection with 4 to 6-way interactions. (Kuhn R. D., Kacker, Lei, & Simos, 2020)

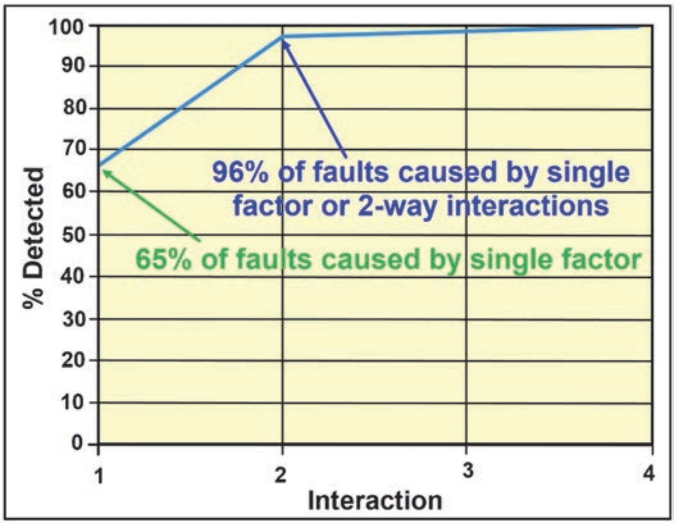


Figure 1-3. How are interaction faults distributed

Matrices known as covering arrays can be computed to cover all t-way combinations of variable values, up to a specified level of t (typically $t \leq 6$), making it possible to efficiently test all such t-way interactions (Kuhn, Wallace, & Gallo, 2004). The effectiveness of any software testing technique depends on whether test settings corresponding to the actual defects in the SUT are included in the test sets. When test sets do not include test settings corresponding to actual defects, the faults and defect will not be detected. Conversely, with the right input model, we have evidence that the software works correctly for t-way combinations contained in passing tests.

1.6 Input Space

Among the key questions in software testing and assurance are what parameters matter, and what values should be included in testing. Except for trivial problems, it is generally intractable to test all possible inputs, so some form of equivalence partitioning must be used, i.e., the input space is divided into sets of inputs that are considered equivalent with respect to some relation that is meaningful for the software under test. Put simply, the problem is to find parameter values that each adequately represent a much larger set of values. A simple example is partitioning into valid and invalid input values. Another example might be ranges of weights or sizes that are equivalent with respect to the shipping charge for each class.

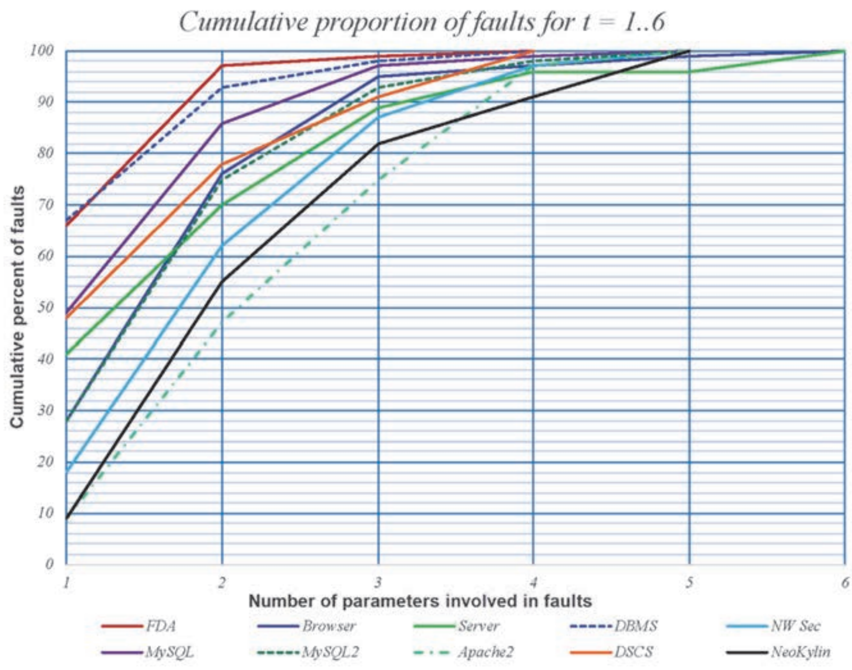


Figure 1-4. Cumulative fault distribution

Because the input space is far too large to test exhaustively, inputs must be discretized for continuous-valued variables, or a small subset selected from enumerated values. This problem is fundamental in software assurance, and an extensive body of research has been developed to solve it. In general, input parameters cannot be considered in isolation, because the value of one may limit the values that must be considered in testing for other parameters, that is, there may be constraints among parameters.

How can an adequate input model be found? Part of the problem was solved decades ago, with systematic methods of analyzing and partitioning the input space. Accepted practices for this task include boundary value analysis (Beizer, 2003), to identify boundaries in range-defined variables and divide the input space into partitions of values for which the system under test can be expected to produce equivalent results. The classic category partition method (Ostrand & Balcer, 1988) provides a systematic way to integrate the definition of boundary values and equivalence partitions into test frames that consolidate values into sets that exercise particular functionality in the SUT. A more recently developed approach that builds on these ideas is the classification tree method (Grochtmann & Grimm, 1993), which adds a graphical notation and analysis of hierarchical or implicit dependencies among parameter values. Such methods are valuable in providing a systematic, rigorous method of input parameter model definition, but a good deal of engineering judgment is still required to determine whether the model is adequate. This partition of the parameters and values is referred to as an input parameter model (IPM), or simply input model.

After an input model has been constructed, how can one ensure that it is adequate for testing, that is, the equivalence partitions contain values that are truly equivalent in terms of system response? More generally, can we find measures of the IPM that are relevant to testing? To answer this question, it is useful to first consider empirical data on the distribution of faults.

1.7 Coverage Implications of Fault Distribution

The empirical distribution of faults suggests that it is useful to consider the degree to which combinations of input values are covered in a test set. The data in ***Figure 1-4*** suggest that we may miss 5-10% of possible errors if we do not test 4-way, or more complex, combinations. However, these findings also mean that in general it will not be necessary to test all possible combinations of input values, because relatively few factors are involved in failures. Covering two-way combinations of inputs, for small values of t, is to some extent equivalent or at least close to exhaustive testing. Thus, it is important to understand the coverage of input combinations for any test set.

As noted, a covering array may be constructed to cover all t-way combinations of input parameters, but any test set of course contains many combinations. We can measure the combinatorial coverage, i.e., the coverage of t-way combinations in a test set, for a better understanding of test set quality. These measures provide quantitative levels of quality very different from conventional structural coverage. In particular, combinatorial coverage has a direct relationship with fault detection. As shown in ***Figure 1-4***, a significant portion of fault triggering combinations involve more than two factors, so the level of combination coverage measures the ability of the test set to detect, for example, faults induced by 4-way or 5-way combinations. These measures can also be computed independently of structural coverage, prior to running any tests, because they relate to the (static) content of the test set. However, there is an interesting relationship between combinatorial coverage and structural coverage, as discussed later.

1.8 Measuring Coverage of Fault-triggering Combinations

Measuring combination coverage can help in understanding the degree of risk that remains after testing. If a high level of coverage of input state-space variable combinations has been achieved, then the risk is small that there is latent combination that may induce a failure. Lower coverage reflects greater risk that there is some untested failure-triggering combination. A variety of measures of combinatorial coverage can help in estimating this risk. In this section we describe some of the basics; see, (Kuhn, Dominguez, Kacker, & Lei, 2013) and for more details.

For a set of t variables, a variable-value configuration is a set of t valid values, one for each of the variables, i.e., the variable-value configuration is a particular setting of the variables. For example, four binary variables a, b, c, and d, for a selection of three variables a, c, and d the set $\{a = 0, c = 1, d = 0\}$ is a variable-value configuration, and the set $\{a = 1, c = 1, d = 0\}$ is a different variable-value configuration.

Table 1-2. Test array with four binary components

a	b	c	d
0	0	0	0
0	1	1	0
1	0	0	1
0	1	1	1

Table 1-3. The test array covers all possible 2-way combinations of a, b, c, and d to different levels

Vars	Configurations covered	Config coverage
a b	00, 01, 10	.75
a c	00, 01, 10	.75
a d	00, 01, 11	.75
b c	00, 11	.50
b d	00, 01, 10, 11	1.0
c d	00, 01, 10, 11	1.0

It is also useful to measure the number of t-way combinations covered out of all possible settings of t variables. For a given combination of t variables, total variable-value configuration coverage is the proportion of all t-way variable-value configurations that are covered by at least one test case in a test set. This measure may also be referred to as total t-way coverage. For the array in ***Table 1-2***, there are 4-choose-2, $C(4{,}2) = 6$ possible variable combinations and $22 \times C(4{,}2) = 24$ possible variable-value configurations. Of these, 19 variable-value configurations are covered and the only ones missing are $ab = 11$, $ac = 11$, $ad = 10$, $bc = 01$, $bc = 10$, so the total variablevalue configuration coverage is 19/24 = 79%. Although the example in ***Table 1-2*** uses variables with the same number of values, it is also possible to compute coverage for test sets in which parameters have differing numbers of values.

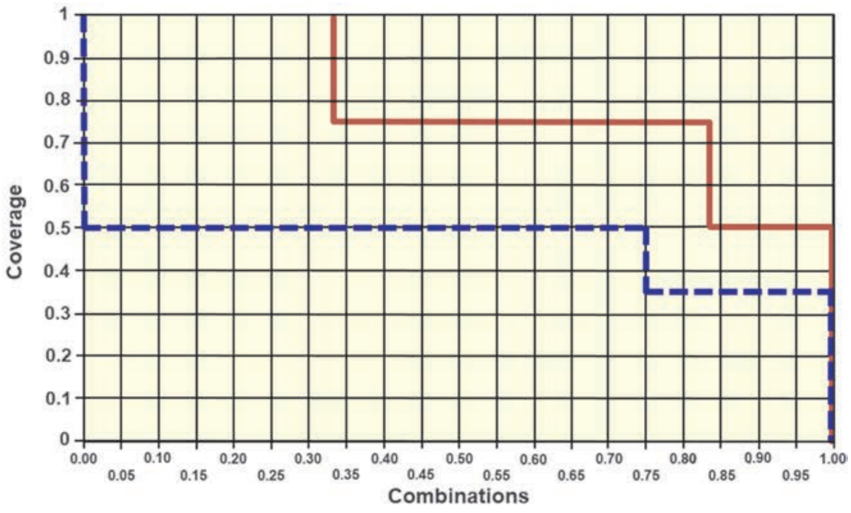


Figure 1-5. Graph of coverage for tests in Table 1-2

Figure 1-5 shows a graph of the 2-way (red/solid) and 3-way (blue/dashed) coverage data for the tests in ***Table 1-2***. Combination coverage of variable values is given as the y-axis, with the percentage of combinations reaching a particular coverage level as the x-axis. Of particular interest is the minimum t-way coverage, M_t, which is the smallest proportion of coverage of variable-value configurations for parameters taken t at a time. For example, ***Table 1-2*** shows four binary variables, a, b, c, and d, where each row represents a test, so parameters taken two at a time have four possible configurations: $00, 01, 10, 11$. The six possible 2-way variable combinations, ab, ac, ad, bc, bd, cd, only bd and cd are covered to different levels, shown in the second column. The minimum coverage for 2-way combinations, $M_2 = 0.5$, shown as $y = 0.5$ at $x = 0.833$ because one out of the six variable combinations has 2 of the 4 possible settings of two binary variables covered. The area under the curve for 2-way combinations is approximately 79% of the total area of the graph, reflecting the total coverage of 19 out of 24 2-way combinations.

1.9 Practical Example

Combinatorial coverage measures were originally developed to analyze the input space coverage of tests for spacecraft control software (Maximoff, Kuhn, Trela, & Kacker, 2010). A test suite of 7,489 tests had been developed using conventional techniques such as use case analysis and knowledge of likely error sources. The control system included 82 variables, with a test configuration of 132754262 (three 1-value, 75 binary, two 4-value, and two 6-value). Figure 3 shows combinatorial coverage achieved by the full set of 7,489 tests (red = 2-way, blue = 3-way, green = 4-way, orange = 5- way). The area under the curves is the proportion of the input space tested at the various t-way levels, while the are above the curve represents untested space – where errors might still be found. For example, 2-way coverage is 94%, so there is relatively little risk of faults that might be triggered by 2-way combinations.

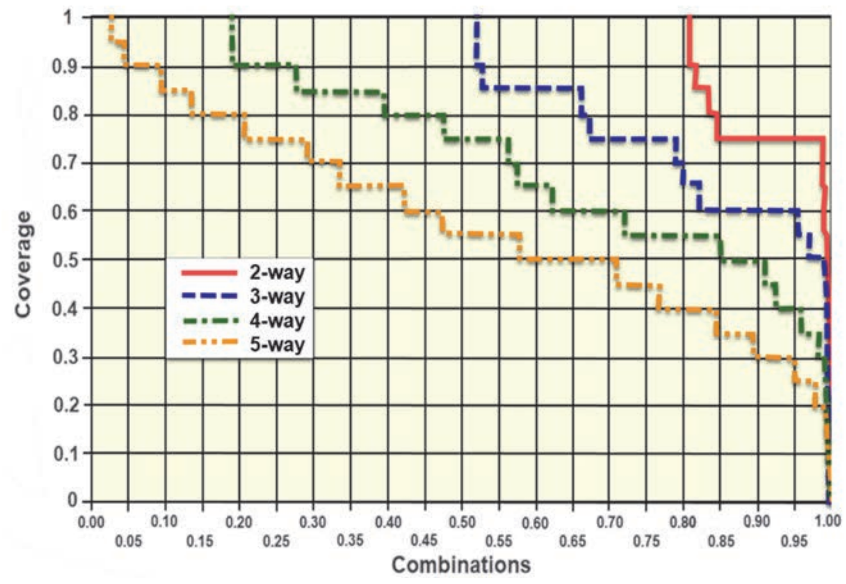


Table 1-4. **Total t-way coverage for Fig. 3 configuration.**

interaction	combinations	settings	coverage
2-way	3321	14761	94.0
3-way	88560	828135	83.1
4-way	1749060	34364130	68.8
5-way	27285336	603068813	53.6

1.10 Relationship Between Combinatorial Coverage and Structural Coverage

It is obvious that any thorough assessment of a system must show that all requirements have been met, but it is also true that the system should not produce unexpected behavior. One of the most effective means of confirming these objectives is to generate tests from requirements, then ensure that full structural coverage has been achieved, for some strong coverage criterion. That is, tests must be requirements-based; structural coverage is used only to validate the quality of these tests. This approach has been required by the US Federal Aviation Administration for testing life-critical aviation software, in RTCA standard DO-178B (RTCA, 1992), and continued in the update DO-178C (Ferrell & Ferrell, 2017). This requirement uses a stronger relative of branch/decision coverage called modified condition decision coverage (MCDC), which subsumes branch coverage. (Branch coverage requires that each branch of every control structure, such as if or while conditionals, has been taken in the test suite. MCDC requires that every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to independently affect the decision outcome, and that each entry and exit point have been invoked at least once.). The intuition is clear – if full structural coverage has not been achieved, then either tests are inadequate, or there is some system functionality that is outside of the specification

1.11 Assured autonomy and AI/ML verification

Input space coverage measurements are needed in assurance and verification of autonomous systems because current methods for assurance of safety critical systems rely on measures of structural coverage, which do not apply to many autonomous systems. Combinatorial methods, including a theorem relating measures of input space coverage, offer a better approach for autonomous system verification and AI assurance.

1.12 Coverage Measures

A complete input model is part of the goal of achieving thorough testing. Depending on what system aspects are to be considered in defining completeness, a variety of approaches exist to determine when testing is considered enough. Typically, these include some notion of fully covering requirements, and may also consider structural coverage of the code. In software engineering, structural coverage refers to measures of the degree to which programs have been exercised. Two of the most widely used measures are statement coverage, the proportion of program statements that have been executed, and branch coverage (also known as decision coverage), the proportion of branches that have been evaluated to both true and false. Many other measures or test criteria exist, including condition coverage and modified condition decision coverage, and it can be shown that these criteria form a hierarchy (Ammann & Offutt, 2016).

For example, decision coverage subsumes statement coverage. Structural coverage measures are of value in gauging the thoroughness of a test set, although their utility is somewhat limited. Statement coverage is the weakest of these measures, but failure to achieve full statement coverage at least indicates that code has not been tested well enough. Branch coverage provides a stronger measure, and more complex criteria related to branch coverage are used for some life critical applications such as aviation.

Of course, simply achieving some level of structural coverage is hardly adequate because it must be shown that the SUT accomplishes its intended function. If the code does only one of ten required functions, the level of structural coverage is obviously an inadequate measure for determining if all of the ten functions have been implemented. The SUT may work well for the subset of inputs related to the implemented function, but it would fail if presented with inputs directing execution of unimplemented functions. Similarly, if all of the functions are implemented but not all correct, the code may produce the right result for some input values but fail on others.

Responsible test engineers know the importance of test data in the development of an efficient test automation framework. Well managed test data in a framework is built around the strategy that there is high test coverage as a result. But, what if there are many test data inputs considered? What if we do not have time to test the several combinations of test data inputs in the process of manual or Automation testing? How can we test fewer test cases and still test “right”?

Well, this is when combinatorial test design can help. It helps design an optimized set of test cases (based on the set of test data variables) that ensure 100% test coverage. The high-quality test design CTD result, in turn, can be fed as input to the automation testing tools, thus speeding up the delivery in the Quality Analysis stage.

Another challenge to software test design is automation. Automation is one of the more popular and available strategies to reduce testing effort. It develops test scripts that will be used later to execute test cases instead of human (Wissink & Amaro, 2006). The idea behind automation is to let computer simulate what the tester is doing in reality when running test cases manually on SUT. Top challenges with test automation include:

- Slow Test Authoring
- Poor environment fit
- High test maintenance
- Slow troubleshooting
- Fast execution
- Skills match

Automation is one of the more popular and available strategies to reduce testing effort. It develops test scripts that will be used later to execute test cases instead of human (Wissink & Amaro, 2006). The idea behind automation is to let computer simulate what the tester is doing in reality when running test cases manually on SUT.

1.13 CDT - The mathematical background behind CTD

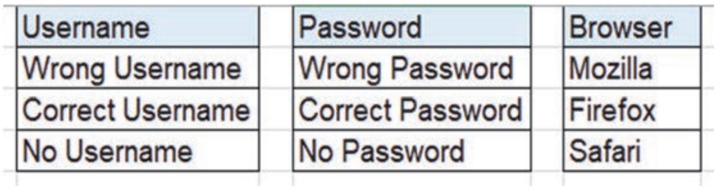
CTD is a technique that helps plan test data as input to the manual or automation test process, ensuring 100% test coverage based on the several combinations of test data inputs for a test suite. CTD form a crucial branch of combinatorics, a subdomain of Discrete Mathematics. The mathematical concept of "Orthogonal Arrays" sits at the base of the combinatorial test design. A mathematician named Taguchi coined the mathematical algorithm term, Orthogonal Arrays (OA). CTD encompasses a wide range of concepts, including but not limited to **block designs**, **t-designs**, and **Latin squares**.

1.14 Strengthening existing data-driven test techniques

Consider the scenario where you have test data stored in an excel file. You may be testing the application manually or using test automation tools. In this case, you can feed the test data inputs to the CTD tool to help you generate a combination of the test data input interactions. This improved set of test data goes as an input to the manual test or automation test processes. Many researchers use the OA concept for their industrial experiments in many domains all over the world.

1.15 *Examples of CTD scenarios*

For example, let us say there is a simple test to validate the login functionality of an application on several types of browsers and devices. Here are the variables: Username, password, and browser. Please note the values of the variables in the following table:



Username	Password	Browser
Wrong Username	Wrong Password	Mozilla
Correct Username	Correct Password	Firefox
No Username	No Password	Safari

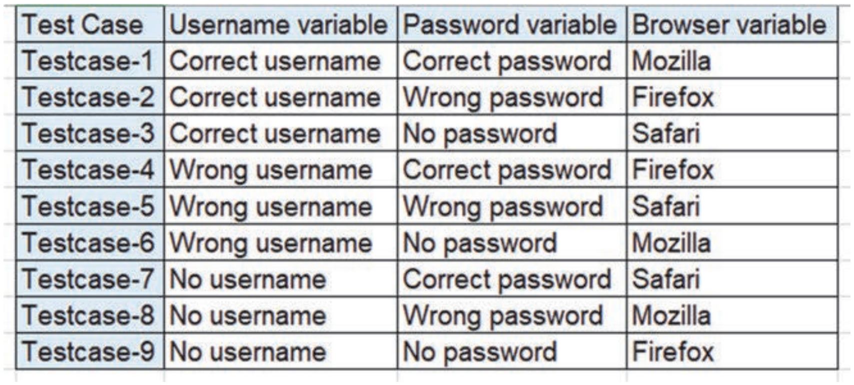
Figure 1-6. Table of variables: Username, password, and browser

In the above case, in case you plan to test all the combinations of tests, in a multi-dimensional pattern, it would take 27 tests to design a complete coverage of the tests!

$$3 \times 3 \times 3 = 27 \text{ tests}$$

However, In the case of CTD, if you use a tool that is based on the OA concept, it helps you derive tests based on N-pair interaction of variables, at the same time ensuring full coverage, with a few numbers of tests! The N-wise dimension of testing aims to test all the possibilities of all random combination of N variables' values.

The following displayed table is the result of a two-pair interaction of variable values. There are only nine test cases in the outcome! The CTD result lists the final test cases, eventually finding a small test plan that covers complete coverage. Note that, two-pair means that every variable found will be paired with another one variable in a two-pair set.



Test Case	Username variable	Password variable	Browser variable
Testcase-1	Correct username	Correct password	Mozilla
Testcase-2	Correct username	Wrong password	Firefox
Testcase-3	Correct username	No password	Safari
Testcase-4	Wrong username	Correct password	Firefox
Testcase-5	Wrong username	Wrong password	Safari
Testcase-6	Wrong username	No password	Mozilla
Testcase-7	No username	Correct password	Safari
Testcase-8	No username	Wrong password	Mozilla
Testcase-9	No username	No password	Firefox

Figure 1-7. Two-pair interaction test cases

The above is an Orthogonal Array built from a two-pair interaction of variables. Here, you will notice that every variable is paired with all the other variables at least once!

Example, "No Password" has been covered in a pair at least in one test case with the variables of:

- "Username variable" values which are – correct username, wrong username, No username.
- "Browser variable" values which are – Mozilla, Safari, Firefox.

There are only nine tests that have come out of the Orthogonal Array calculation Versus that all-pair combination value of 27 tests still ensuring full coverage in test design planning! Even then, there is a 100% coverage of tests, with the two-pair interaction levels!

1.16 How CTD aids Agile and DevOps

DevOps encourages Automation in all forms, and this is what a CTD tool precisely delivers by helping the test engineer. It enables the team by using an automated fashion of producing an efficient test design. Using the CTD tool, you can quickly build a set of an optimized set of test cases, which can be served as an input to the test automation scenarios.

Automated tests can provide quick results, and fast feedback compared to a manually planned test design method. It eventually also saves time and avoids long feedback cycles. All of this encourages the Agile way of working. As a result, there is early and continuous delivery of quality products.

Automated tools that help build CTD scenarios.

- IBM FOCUS (https://www.ibm.com/docs/en/rpa/23.0?topic=controls-focus) is a licensed CTD tool used for test planning. It can read the test data inputs, analyze the test data coverage, select a subset of the tests, and generate a new test plan ensuring full range.
- The *pairwise testing tool* (https://www.pairwise.org/) also offers the all-pairs solution for test design solutions, presenting a visualized test coverage view to the test engineers.
- Other available tools (see below)

The test scenarios result in that comes as a result of the above-mentioned tools can be in turn fed as an input to the test automation tools, thus speeding up the test design process.

1.17 Wrap Up

The field of combinatorics has come a long way, and CTD is one of the branches that has helped the test teams in many ways by assisting them in building efficient test designs for the product. No coding skills are required to construct a CTD design. The tabular result is easy to understand. Hence, this helps in aiding the communication between the test and business analyst teams while discussing the software system requirements. A mathematical method as such should be heavily relied on considering that it has a high ROI by helping to save time and help to find high-quality defects. Especially in the case of large-scale projects involving a large number of variables to be depended on for testing, CTD can help the organization a lot.

2 GOAL OF CDT IN SOFTWARE TESTING

The main goal of **combinatorial testing** is to make sure that the software product can handle different combinations of test data as input parameters and configuration options. Testers use **combinatorial testing** based on the idea that defects in a software application can occur by specific combinations of input parameters rather than by isolated inputs. By focusing on the input combinations like that, the testers can provide effective test coverage while reducing the number of test cases written.

2.1 Benefits of Combinatorial Testing

Combinatorial testing has many advantages when it comes to ensuring the quality of a software product. That is why testers choose **combinatorial testing** over normal software testing methods when testing complicated software applications. Let us get to know some of the benefits of **combinatorial testing**.

- Covers a broad range of input combinations using a minimum number of test cases.
- Increases test coverage compared to normal component testing since it always considers multiple input combinations.
- Helps to detect bugs, defects, vulnerabilities, and unexpected outputs that might not be detected during the usual component and regression testing phases.
- Reduces testing effort, cost, and time. (Since combinatorial tests use fewer test cases to cover a wide scope of testing.)
- Identifies issues at the earliest while allowing the team to address and fix those earlier in the software development life cycle.
- Optimizes the testing process by removing unwanted test cases while ensuring that the cost and effort are not wasted on repeating the same test scenarios again and again.
- Helps to test complex software applications with a large number of parameters, settings, and options.
- Reduces the risk of critical defects going unnoticed, which can occur only when handling specific input combinations.

2.2 Combinatorial Test Case

A combinatorial test case is very different from a normal test case, and it is a specific test scenario that is created using combinatorial test techniques. That is designed to cover various combinations of input parameters of a software application. Let us get to know some key points about combinatorial test cases.

- Each combinatorial test case is designed to cover a specific input parameter of variable combination.
- Combinatorial test cases are created to cover a wide range of the scope since the end goal of **combinatorial testing** is to reduce the number of test cases.
- Combinatorial test cases are not created randomly. They are designed using specialized algorithms or tools that consider various input parameters to create test cases.
- The basic need for combinatorial test cases is to reduce the effort of testing complicated software applications focusing on critical combinations.
- Like the other types of test cases, the testers should be able to detect defects, bugs, vulnerabilities, and unexpected system behaviors when combinatorial test cases are executed.

Let's take a **combinatorial test example** for a simple login scenario. Initially, a login page of a web application might contain parameters such as user name and password. Those parameters can have several inputs like this,

- Username: [User1, User2, User3]
- Password: [Password1, Password2]

If we consider this login scenario to write combinatorial test cases, we can design 6 possible combinatorial test cases for that. Those will be like the following,

- Test Case 1: Username=User1, Password=Password1
- Test Case 2: Username=User2, Password=Password1
- Test Case 3: Username=User3, Password=Password1
- Test Case 4: Username=User1, Password=Password2
- Test Case 5: Username=User2, Password=Password2

- Test Case 6: Username=User3, Password=Password2

In this **combinatorial test example**, we can cover all the possible input combinations of usernames and passwords using only 6 test cases. With that, now you have an idea of how to write a minimum number of test cases to cover a broad range of the testing scope.

2.3 How to Perform Combinatorial Testing

Combinatorial testing cannot be done with the same approach that we use to perform normal component testing since it always focuses on the input parameter combinations rather than testing the initial scenario. To perform a combinatorial test, the following steps can be followed.

1. Identify input parameters and configuration options that need to be tested in the software application, including variables, settings, features, etc.
2. For each of these parameters or the configurations, define a list of possible values that represent different settings and options. (Such as different usernames, passwords, etc.)
3. Insert the list of possible values into the selected **combinatorial testing** tool and design the respective combinatorial test cases. Or create the combinatorial test cases manually considering the respective input combinations.
4. Execute the created test cases on your software application.
5. Analyze the results of the test execution and review the combinatorial test coverage.
6. Repeat the testing process with various input parameters and configurations if you need to.
7. Report any issues found during combinatorial test execution.
8. Continue maintenance testing by re-running combinatorial tests from time to time with necessary changes.

2.4 Manual Combinatorial Testing

When it comes to manual **combinatorial testing**, testers manually select and combine input parameters and their values to cover various combinations. They need to plan, design, and execute combinatorial test cases while documenting the test results as well. Even though its control is with the testers, creating combinatorial test cases will be more time-consuming and it will not be very practical for the software applications with a large number of input parameters. This approach is more effective for smaller software applications with a smaller number of input parameter combinations that the automated tools are not suitable for.

2.5 Automated Combinatorial Testing

In automated **combinatorial testing**, testers use specialized software tools or scripts to create and execute a large number of test cases. These automated tools use combinatorial algorithms to create test cases while considering various combinations of input parameters and configuration options. Ultimately, this approach minimizes the time and effort of manual **combinatorial testing,** and it is more effective for software applications with a large number of input parameters.

2.6 Combinatorial Testing Tools

Combinatorial testing tools are very important due to several reasons. Let’s take a software application that needs to be tested with a lot of input variables. Other than performing combinatorial testing manually, we can get help from the combinatorial testing tools which create a minimum number of test cases that cover a wide range of input combinations. It reduces the time and effort required for manual **combinatorial testing** as well. When it comes to accuracy, we can trust the test cases that were written by an automated approach rather than the manually written ones.

Combinatorial testing tools help detect defects, vulnerabilities, and unexpected responses effectively. Most importantly, **combinatorial testing tools** can be effectively used when testing more complex software applications instead of doing **combinatorial testing** manually. Because, if testers create and execute combinatorial test cases manually for a more complex software application, there is a high chance of missing several critical test scenarios that can lead the entire software product to a high risk.

2.7 Top Combinatorial Testing Tools

Even though **combinatorial testing** sounds like quite a new concept, several tools can be used to perform **combinatorial testing** which are currently popular among quality assurance engineers. Here are some of those tools.

- Advanced Combinatorial Testing System (ACTS) <https://www.afit.edu/STAT/statcoe_files/Combinatorial%2520Test%2520Designs%2520Final.pdf >
- Pairwise Independent Combinatorial Testing (PICT) <https://github.com/microsoft/pict>

2.8 Advantages of Combinatorial Testing

Combinatorial testing has a lot of advantages that make the tester's life easier. Let's get to know some of those advantages.

- **Combinatorial testing** provides a wide test coverage with a smaller number of test cases which verifies the software application considering various input combinations.
- **Combinatorial testing** reduces testing effort, time, and cost, especially for software applications with a large number of input variables.
- This testing method is highly effective when it comes to detecting issues that occur due to specific input combinations.
- **Combinatorial testing** helps to detect issues during the earlier stages of development while reducing the cost of fixing them in later development phases.

2.9 Disadvantages of Combinatorial Testing

Even though there are plenty of advantages that the software development teams can gain from using **combinatorial testing** when ensuring the quality of the software applications, there is a set of disadvantages and challenges in **combinatorial testing** as well. Let's get to know some of those disadvantages and challenges.

- **Combinatorial testing** may become more complex when dealing with a huge number of input parameters and values. It will be a bit challenging for the testers to manage those combinations manually.
- Since the testers may need to use specialized tools to perform automated **combinatorial testing**, it will take considerable time and effort for the testers to dive into those new tools.
- Automated **combinatorial testing tools** may require more computing resources and it will be challenging for the software projects which function under a smaller number of resources.
- **Combinatorial testing** focuses on specific input combinations and sometimes they may not represent real-world user behavior.
- With the changes that are being added to the particular software application, combinatorial test cases should be updated and well-maintained from time to time.

2.10 Conclusion

Combinatorial testing helps the testers to effectively verify the behaviors of a software application by designing and executing test cases considering various combinations of input variables and configurations. This testing approach is very beneficial especially when it comes to software applications with a huge complexity. Ultimately, **combinatorial testing** covers a wide scope of testing with a smaller number of test cases while reducing the cost, time, and effort. Even though there are a few disadvantages, with the usage of the most suitable tools and adhering to the correct **combinatorial test design** steps, the testers can tackle those challenges and get the maximum out of **combinatorial testing**. All in all, **combinatorial testing** is the best way of giving the maximum test coverage with the minimum number of test cases.

3 TEST GENERATION FROM COMBINATORIAL DESIGNS

The purpose of this chapter is to introduce techniques for the generation of test configurations and test data using the combinatorial design techniques with program inputs and their values as, respectively, factors and levels. These techniques are useful when testing a variety of applications. They allow selection of a small set of test configurations from an often impractically large set and are effective in detecting faults arising out of factor interactions.

3.1 Combinatorial Designs

Software applications are often designed to work in a variety of environments. Combinations of factors such as the operating system, network connection, and hardware platform, lead to a variety of environments. Each environment corresponds to a given set of values for each factor, known as a *test configuration*. For example, Windows XP, Dial-up connection, and a PC with 512 MB of main memory, is one possible configuration. To ensure high reliability across the intended environments, the application must be tested under as many test configurations, or environments, as possible. However, as illustrated in examples later in this chapter, the number of such test configurations could be exorbitantly large making it impossible to test the application exhaustively.

An analogous situation arises in the testing of programs that have one or more input variables. Each test run of a program often requires at least one value for each variable. For example, a program to find the greatest common divisor of two integers x and y requires two values, one corresponding to x and the other to y. In earlier chapters we have seen how program inputs can be selected using techniques such as equivalence partitioning and boundary value analysis. While these techniques offer a set of guidelines to design test cases, they suffer from two shortcomings: (a) they raise the possibility of a large number of subdomains in the partition of the input space and (b) they lack guidelines on how to select inputs from various sub-domains in the partition.

The number of sub-domains in a partition of the input domain increases in direct proportion to the number and type of input variables, and especially so when multidimensional partitioning is used. Also, once a partition is determined, one selects at random a value from each of the sub-domains. Such a selection procedure, especially when using unidimensional equivalence partitioning, does not account for the possibility of faults in the program under test that arise due to specific interactions amongst values of different input variables. While boundary value analysis leads to the selection of test cases that test a program at the boundaries of the input domain, other interactions in the input domain might remain untested.

This chapter describes several techniques for generating test configurations or test sets that are small even when the set of possible configurations, or the input domain, and the number of sub-domains in its partition, is large and complex. The number of test configurations, or the test set so generated, has been found to be effective in the discovery of faults due to the interaction of various input variables. The techniques we describe here are known by several names such as design of experiments, combinatorial designs, orthogonal designs, interaction testing, and pairwise testing.

3.1.1 Test configuration and test set

In this chapter we use the terms *test configuration* and *test set* interchangeably. Even though we use the terms interchangeably, they do have different meaning in the context of software testing. However, the techniques described in this chapter apply the generation of both test configurations as well as test sets, we have taken the liberty of using them interchangeably. One must be aware that a test configuration is usually a static selection of factors, such as the hardware platform or an operating system. Such selection is completed prior to the start of the test. In contrast, a test set is a collection of test cases used as input during the test process.

3.2 Modeling the input and configuration spaces

The input space of a program P consists of k-tuples of values that could be input to P during execution. The configuration space of P consists of all possible settings of the environment variables under which P could be used.

EXAMPLE 3.1. Consider program P that takes two integers $x > 0$ and $y > 0$ as inputs. The input space of P is the set of all pairs of positive non-zero integers. Now suppose that this program is intended to be executed under the Windows and the Mac OS operating system, through the Netscape or Safari browsers, and must be able to print to a local or a networked printer. The configuration space of P consists of triples (X, Y, Z) where X represents an operating system, Y a browser, and Z a local or a networked printer. ■

Next, consider a program P that takes n inputs corresponding to variables $X_1, X_2, \ldots, X_n$. We refer to the inputs as *factors*. The inputs are also referred to as *test parameters* or as *values*. Let us assume that each factor may be set at any one from a total of c_i , $1 \leq i \leq n$ values. Each value assignable to a factor is known as a *level*. The notation $|F|$ refers to the number of levels for factor F .

The environment under which an application is intended to be used generally contributes one or more factors. In Example 12.1, the operating system, browser, and printer connection are three factors that will likely affect the operation and performance of P .

A set of values, one for each factor, is known as a *factor combination*. For example, suppose program P has two input variables X and Y . Let's say that during an execution of P , X and Y may each assume a value from the set $\{a, b, c\}$ and $\{d, e, f\}$, respectively. So, we have 2 factors and 3 levels for each factor. This leads to a total of $32 = 9$ factor combinations, namely (a, d), (a, e), (a, f), (b, d), (b, e), (b, f), (c, d), (c, e), and (c, f). In general, for k factors with each factor assuming a value from a set of n values, the total number of factor combinations is nk .

Suppose now that each factor combination yields one test case. For many programs, the number of tests generated for exhaustive testing could be exorbitantly large. For example, if a program has 15 factors with 4 levels each, the total number of tests is $415 \approx 109$. Executing a billion tests might be impractical for many software applications.

There are special combinatorial design techniques that enable the selection of a small subset of factor combinations from the complete set. This sample is targeted at specific types of faults known as *interaction* faults. Before we describe how the combinatorial designs are generated, let us look at a few examples that illustrate where they are useful.

EXAMPLE 3.2. Let's model the input space of an online Pizza Delivery Service (PDS) for the purpose of testing. The service takes orders online, checks for their validity, and schedules Pizza for delivery. A customer is required to specify the following four items as part of the online order: Pizza size, Toppings list, Delivery address, and a home phone number. We'll denote these four factors by S, T, A, and P, respectively.

Suppose there are three varieties for size: Large, Medium, and Small. There is a list of 6 toppings from which to select. In addition, the customer can customize the toppings. The delivery address consists of customer name, one line of address, city, and the zip code. The phone number is a numeric string possibly containing the dash ("–") separator.

Table 3-1 lists one model of the input space for the PDS. Note that while for Size we have selected all three possible levels, we have constrained the other factors to a smaller set of levels. Thus we are concerned with only one of two types of values for Toppings, Custom or Preset, and one of the two types of values for factors Address and Phone, namely Valid and Invalid.

Table 3-1. One model of the input space for the PDS

Factor	Levels		
Size	Large	Medium	Small

Toppings	Custom	Preset
Address	Valid	Invalid
Phone	Valid	Invalid

The total number of factor combinations is $24 + 23 \; = 24$. However, as an alternate to the ***Table 3-1*** above, we could consider $6 + 1 = 7$ levels for Toppings. This would increase the number of combinations to $24 + 5 \times 23 + 23 + 5 \times 22 \; = \; 84$. We could also consider additional types of values for Address and Phone which would further increase the number of distinct combinations. Notice that if we were to consider each possible valid and invalid string of characters, limited only by length, as a level for Address, we will arrive at a huge number of factor combinations.

Table 3-2. Factors and levels for the Unix `sort` *utility.*

Factor	Meaning	Levels			
-	Forces the source to be the standard input.	Unused	Used		
-c	Verify that the input is sorted according to the options specified on the command line.	Unused	Used		
-m	Merge sorted input	Unused	Used		
-u	Suppress all but one of the matching keys.	Unused	Used		
-o output	Output sent to a file.	Unused	Valid file	Invalid file	
-Tdirectory	Temporary directory for sorting.	Unused	Exists	Does not exist	
-ykmem	Use *kmem* kilobytes of memory for sorting.	Unused	Valid *kmem*	Invalid *kmem*	
-zrecsize	Specfies record size to hold each line from the input file.	Unused	Zero size	Large size	
-dfiMnr	Perform dictionary sort	Unused	fi	Mnr	fiMnr

Later in this section we explain the advantages and disadvantages of limiting the number of factor combinations by partitioning the set of values for each factor into a few subsets. Also notice the similarity of this approach with equivalence partitioning. The next example illustrates factors in a GUI.

EXAMPLE 3.3. The Graphical User Interface (GUI) of application T consists of three menus labeled File, Edit, and Format. Each menu contains several items listed in ***Table 3-3*** below.

Table 3-3.GUI Factors and Levels

Factor			Levels	
File	New	Open	Save	Close
Edit	Cut	Copy	Paste	Select
Typeset	LaTex	BibTex	PlainTeX	MakeIndex

We have three factors in T . Each of these three factors can be set to any of four levels. Thus we have a total $43 = 64$ factor combinations.

Note that each factor in this example corresponds to a relatively smaller set of levels when compared to the factors Address and Phone in the previous example. Hence the number of levels for each factor is set equal to the cardinality of the set of the corresponding values.

EXAMPLE 3.4. Let's consider the Unix `sort` utility for sorting ASCII data in files or obtained from the standard input. The utility has several options and makes an interesting example for the identification of factors and levels. The command line for `sort` is as given below.

```
sort [ -cmu ] [ -o output ] [ -T directory ] [ -y
[ kmem ]] [ -z recsz ] [ -dfiMnr ] [ - b ] [ t
char ] [ -k keydef ] [ +pos1 [ -pos2 ]] [ file...]
```

Table 3-4. Factors and levels for the Unix sort utility (continued).

Factor	Meaning	Levels	
-f	Ignore case.	Unused	Used
-i	Ignore non-ASCII characters.	Unused	Used
-M	Fields are compared as months.	Unused	Used
-n	Sort input numerically.	Unused	Used
-r	Reverse the output order.	Unused	Used
-b	Ignore leading blanks	Unused	Used

Factor	Meaning	Levels			
	when using +pos1 and -pos2.				
-tc	Use character *c* as field separator.	Unused	c_1	$c_1 c_2$	
-kkeydef	Restricted sort key definition.	Unused	start	end	starttype
+pos1	Start position in the input line for comparing fields.	Unused	f.c	f	0.c
-pos2	End position for comparing fields.	Unused	f.c	f	0.c
file	File to be sorted.	Not specified	Exists	Does not exist	

Table 3-2 and ***Table 3-4*** list all the factors of sort and their corresponding levels. Note that the levels have been derived using equivalence partitioning for each option and are not unique. We have decided to limit the number of levels for each factor to 4. You could come up with a different, and possibly a larger or a smaller, set of levels for each factor.

In ***Table 3-2*** level *Unused* indicates that the corresponding option is not used while testing the sort command. *Used* means that the option is used. Level *Valid File* indicates that the file specified using the -o option exists whereas *Invalid File* indicates that the specified file does not exist. Other options can be interpreted similarly.

We have identified a total of 20 factors for the sort command. The levels listed in ***Table 3-4*** lead to a total of approximately 1.9×10^9 combinations.

EXAMPLE 3.5. There is often a need to test a web application on different platforms to ensure that any claim such as "Application X can be used under Windows and OS X." Here we consider a combination of hardware, operating system, and a browser as a platform. Such testing is commonly referred to as *compatibility* testing.

Let us identify factors and levels needed in the compatibility testing of application X. Given that we would like X to work on a variety of hardware, OS, and browser combinations, it is easy to obtain three factors, i.e. hardware, OS, and browser. These are listed in the top row of Table 12.3. Notice that instead of listing factors in different rows, we now list them in different columns. The levels for each factor are listed in rows under the corresponding columns. This has been done to simply the formatting of the table.

A quick examination of the factors and levels in ***Table 3-3*** reveals that there are 75 factor combinations. However, some of these combinations are infeasible. For example, OS 10.2 is an OS for the Apple computers and not for the Dell Dimension series PCs. Similarly, the Safari browser is used on Apple computers and not on the PC in the Dell Series. While various editions of the Windows OS can be used on an Apple computer using an OS bridge such as the Virtual PC or Boot Camp, we assume that this is not the case for testing application X.

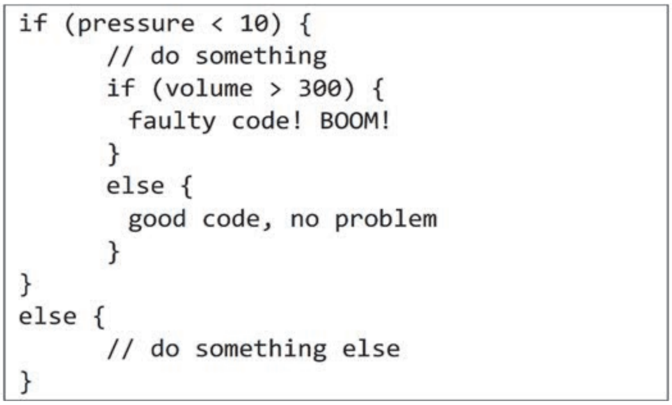
4 COMBINATORIAL METHODS IN TESTING

Developers of large data-intensive software often notice an interesting—though not surprising—phenomenon: When usage of an application jumps dramatically, components that have operated for months without trouble suddenly develop previously undetected errors. For example, the application may have been installed on a different OS-hardware-DBMS-networking platform, or newly added customers may have account records with an oddball combination of values that have not occurred before. Some of these rare combinations trigger failures that have escaped previous testing and extensive use. Such failures are known as interaction failures because they are only exposed when two or more input values interact to cause the program to reach an incorrect result.

Combinatorial testing can help detect problems like this early in the testing life cycle. The key insight underlying *t*-way combinatorial testing is that not every parameter contributes to every failure and most failures are triggered by a single parameter value or interactions between a relatively small number of parameters (for more on the number of parameters interacting in failures, see Appendix B). To detect interaction failures, software developers often use "pairwise testing", in which all possible pairs of parameter values are covered by at least one test. Its effectiveness is based on the observation that software failures often involve interactions between parameters. For example, a router may be observed to fail only for a particular protocol when packet volume exceeds a certain rate, a 2-way interaction between protocol type and packet rate. ***Figure 4-1*** illustrates how such a 2- way interaction may happen in code. Note that the failure will only be triggered when both pressure < 10 and volume > 300 are true. (Kuhn, Kacker, & Lei, 2010)

Pairwise testing can be highly effective and good tools are available to generate arrays with all pairs of parameter value combinations. But until recently only a handful of tools could generate combinations beyond 2-way, and most that did so could require impractically long times to generate 3-way, 4-way, or 5-way arrays because the generation process is mathematically complex. Pairwise testing, i.e. 2-

way combinations, has come to be accepted as the common approach to combinatorial testing because it is computationally tractable and reasonably effective.



```
if (pressure < 10) {
      // do something
      if (volume > 300) {
        faulty code! BOOM!
      }
      else {
        good code, no problem
      }
}
else {
      // do something else
}
```

Figure 4-1. 2-way interaction failure triggered only when two conditions are true.

But what if some failure is triggered only by a very unusual combination of 3, 4, or more sensor values? It is very unlikely that pairwise tests would detect this unusual case; we would need to test 3-way and 4-way combinations of values. But is testing all 4-way combinations enough to detect all errors? What degree of interaction occurs in real failures in errors? Surprisingly, this question had not been studied when NIST began investigating interaction failures in 1999. Results showed that across a variety of domains, all failures could be triggered by a maximum of 4-way to 6-way interactions [34, 35, 36, 65]. As shown in ***Figure 4-2***, the detection rate increased rapidly with interaction strength (the interaction level t in t-way combinations is often referred to as *strength*). With the NASA application, for example, 67% of the failures were triggered by only a single parameter value, 93% by 2-way combinations, and 98% by 3-way combinations. The detection rate curves for the other applications studied are similar, reaching 100% detection with 4 to 6way interactions. Studies by other researchers [6, 7, 26] have been consistent with these results. (Hagar, Wissink, Kuhn, & Kacker, 2015)

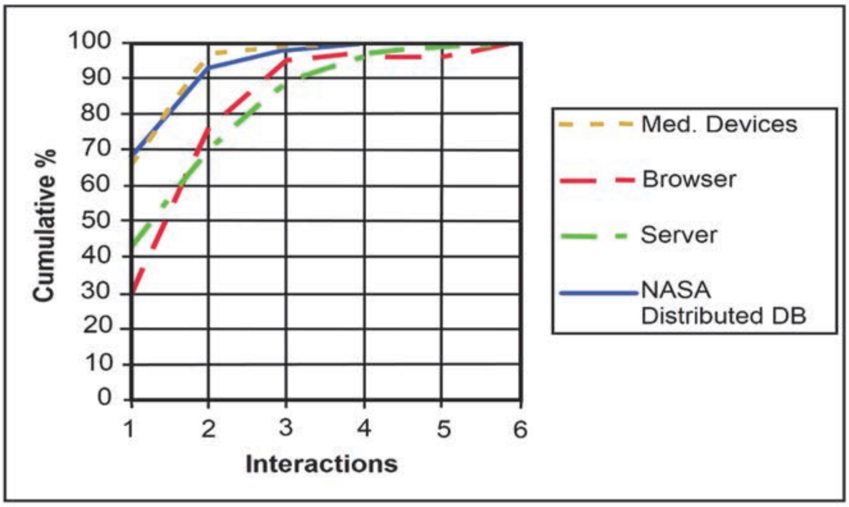


Figure 4-2. ***Error detection rates for interaction strengths 1 to 6***

While not conclusive, these results are interesting because they suggest that, while pairwise testing is not sufficient, the degree of interaction involved in failures is relatively low. We summarize this result in what we call the *interaction rule*, an empirically-derived rule that characterizes the distribution of interaction faults:

Interaction Rule: ***Most failures are induced by single factor faults or by the joint combinatorial effect (interaction) of two factors, with progressively fewer failures induced by interactions between three or more factors.***

Testing all 4-way to 6-way combinations may therefore provide reasonably high assurance. As with most issues in software, however, the situation is not that simple. Efficient generation of test suites to cover all *t*-way combinations is a difficult mathematical problem that has been studied for nearly a century. In addition, most parameters are continuous variables which have possible values in a very large range (+/2^{32} or more). These values must be discretized to a few distinct values. Most glaring of all is the problem of determining the correct result that should be expected from the system under test for each set of test inputs. Generating 1,000 test data inputs is of little help if we cannot determine what the system under test (SUT) should produce as output for each of the 1,000 tests.

With the exception of combination covering test generation, these challenges are common to all types of software testing, and a variety of good techniques have been software testing, and a variety of good techniques have been developed for dealing with them. What has made combinatorial testing practical today is the development of efficient algorithms combinatorial to generate tests covering t-way combinations, and effective testing beyond methods of integrating the tests produced into the testing process. A variety of approaches introduced in this publication can be used to make combinatorial testing a practical and effective addition to the software tester's toolbox.

A note on terminology: we use the definitions below, following the Institute of Electrical and Electronics Engineers (IEEE). The term "bug" may also be used where its meaning is clear.

- ***error***: a mistake made by a developer. This could be a coding error or a misunderstanding of requirements or specification.
- ***fault***: a difference between an incorrect program and one that correctly implements a specification. An error may result in one or more faults.
- ***failure***: a result that differs from the correct result as specified. A fault in code may result in zero or more failures, depending on inputs and execution path.

4.1 Two Forms of Combinatorial Testing

There are basically two approaches to combinatorial testing–use combinations of configuration parameter values, or combinations of input parameter values. In the first case, we select combinations of values of configurable parameters. For example, a server might be tested by setting up all 4-way combinations of configuration parameters such as number of simultaneous connections allowed, memory, OS, database size, etc., with the same test suite run against each configuration. The tests may have been constructed using any methodology, not necessarily combinatorial coverage. The combinatorial aspect of this approach is in achieving combinatorial coverage of configuration parameter values. (Note, the term variable is often used interchangeably with parameter to refer to inputs to a function.)

In the second approach, we select combinations of *input data* values, which then become part of complete test cases, creating a test suite for the application. In this case combinatorial coverage of input data values is required for tests constructed. A typical ad hoc approach to testing involves subject matter experts setting up use scenarios, then selecting input values to exercise the application in each scenario, possibly supplementing these tests with unusual or suspected problem cases. In the combinatorial approach to input data selection, a test data generation tool is used to cover all combinations of input values up to some specified limit. One such tool is ACTS (described in Chapter 7), which is available freely from NIST.

4.2 Configuration Testing

Many, if not most, software systems have a large number of configuration parameters. Many of the earliest applications of combinatorial testing were in testing all pairs of system configurations. For example, telecommunications software may be configured to work with different types of call (local, long distance, international), billing (caller, phone card, 800), access (ISDN, VOIP, PBX), and server for billing (Windows Server, Linux/MySQL, Oracle). The software must work correctly with all combinations of these, so a single test suite could be applied to all pairwise combinations of these four major configuration items. Any system with a variety of configuration options is a suitable candidate for this type of testing.

Configuration coverage is perhaps the most developed form of combinatorial testing. It has been used for years with pairwise coverage, particularly for applications that must be shown to work across a variety of combinations of operating systems, databases, and network characteristics. For example, suppose we had an application that is intended to run on a variety of platforms comprised of five components: an operating system (Windows XP, Apple OS X, Red Hat

Enterprise Linux), a browser (Internet Explorer, Firefox), protocol stack (IPv4, IPv6), a processor (Intel, AMD), and a database (MySQL, Sybase, Oracle), a total of $3 \cdot 2 \cdot 2 \cdot 2 \cdot 3 = 72$ possible platforms. With only 10 tests, shown in ***Table 4-1***, it is possible to test every component interacting with every other component at least once, i.e., all possible pairs of platform components are covered.

*Table 4-1. **Pairwise test configurations***

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

4.3 Input Parameter Testing

Even if an application has no configuration options, some form of input will be processed. For example, a word processing application may allow the user to select 10 ways to modify some highlighted text: *subscript, superscript, underline, bold, italic, strikethrough, emboss, shadow, small caps*, or *all caps*. The font-processing function within the application that receives these settings as input must process the input and modify the text on the screen correctly. Most options can be combined, such as bold and small caps, but some are incompatible, such as subscript and superscript.

Thorough testing requires that the font-processing function work correctly for all valid combinations of these input settings. But with 10 binary inputs, there are $2^{10} = 1{,}024$ possible combinations. But the empirical analysis reported above shows that failures appear to involve a small number of parameters, and that testing all 3-way combinations may detect 90% or more of bugs. For a word processing application, testing that detects more than 90% of bugs may be a cost-effective choice, but we need to ensure that all 3way combinations of values are tested. To do this, we create a test suite to cover all 3-way combinations (known as a ***covering array***) [12, 14, 23, 26, 30, 43, 63].

4.4 Covering Arrays (CAs)

A covering array is a mathematical object that can be used for software testing purposes. For another introduction, see Hartman. (Software and Hardware Testing Using Combinatorial Covering Suites, 2006)

4.4.1 Motivation

Software often has many possible configurations, each of which could be faulty. Software testing tries to make sure that no actual error occurs in any such configuration. One possible method to achieve this goal is to simulate all possible configurations and check that no error occurs. However, this is typically impractical because there are too many configurations to check.

There are many ways to approach software testing in a more practical way, but few of them capture the abstraction and generality of simply running a "test and check" procedure on various sample configurations. To make this method more practical, we can realize that testing all possible configurations of a software program can be redundant and it might be possible to achieve an adequate simulation of all of the program's behavior with a much smaller number of carefully chosen configurations.

Covering arrays have application in many research areas, e.g., Phadke (1989) showed its use in industrial processes, Hedayat et al. (1999) referred to their use in medicine and agriculture, Shasha et al. (2001) proposed its use in the biology area, Pérez-Espinosa et al. (2016) and Lekivetz and Morgan (2020) used covering arrays for tuning the parameters of machine learning algorithms; recently, Izquierdo-Marquez et al. (2018) proposed to use it in the design of experiments to create compost that guarantees a certain level of quality. Be that as it may, software testing is the area with more applications of these combinatorial objects.

4.4.2 Definition

The notion of *coverage* is used to quantify how well a set of sample configurations “covers'' the set of all configurations. Intuitively, full coverage means that all configurations are chosen, and no coverage means that none are chosen. Intermediate levels of coverage are defined in terms of a parameter t. There are two additional parameters that represent the total number of configurations: k is the number of variables a configuration needs to specify, and v is the number of possible values each of the k variables can take on. While in practice each variable can take on a different number of values, we only consider the homogeneous case (where all inputs take on the same number of values) on this site. Taking t equal to k will produce full coverage and taking t equal to zero will produce no coverage. To understand what intermediate values of t mean we turn to an example.

We will consider a simple program that takes only four binary variables. Thus, $k = 4$ and $v = 2$. To take $t = 4$ (with $v^t = 2^4 = 16$) would give full coverage, and thus the following set of configurations to test the program on:

```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
```

```
1 1 1 0
1 1 1 1
```

If we look at this list of configurations, we notice that whichever two columns out of the four columns are chosen, all possible pairs of values come up. Specifically, the pairs 00, 01, 10 and 11 all appear in the rows when we look at the first and second columns only, the first and third columns only, etc. This is intuitive because the table fully covers the configurations. However, there are other configurations that might also share this property. Specifically, we can consider the table:

```
K=1   k=2
 0     0    0    0
 0     1    1    1
 1     0    1    1
 1     1    0    1
 1     1    1    0
```

This table of five configurations does indeed have the property that for each pair of columns chosen, all possible configurations of the two inputs appear. This property is called 2-coverage and corresponds to when $t = 2$. By abstracting away from the idea of software testing we see this is just an array of numbers and because it satisfies the 2-coverage property we call it a covering array. We can naturally generalize this to arbitrary t by defining t-way coverage to hold on an array when for every choice of t columns from the k columns, all possible $v^t t$-tuples appear within the rows of the array.

4.4.3 Optimality

Now that the definition of a covering array is put forth, we can seek optimal covering arrays. That is, for fixed t, v and k values, what is the size of the smallest covering array? As k determines the number of columns, this question is really asking what is the smallest number of rows that an array can have while satisfying the t-covering property. The above covering array with five rows is in fact optimal, as set forth by a by Kleitman and Spencer (1973). Except in this special case for $t = 2$ and $v = 2$ it is generally unknown what are the sizes of optimal covering arrays. For example, the following covering array for $t = 3$,

$v = 2$, and $k = 4$ has 9 rows, but in fact 8 are sufficient, as listed in Charlie Colbourn's (2022) covering array tables.

0 0 0 1
0 0 1 0
0 1 0 0
0 1 1 1
1 0 0 0
1 0 1 0
1 1 0 1
1 1 1 0
~~1 0 1 1~~

Table for CAN(3,k,2) for k up to 10000 and graph from Colbourn (Colbourn C. , 2022)

k	N	Source
4	8	orthogonal array
5	10	Derive from strength 4
11	12	Derive from strength 4
12	15	tabu search (Nurmela)
14	16	Sloane
• • •		
10000	76	Power CT11^4,T3c

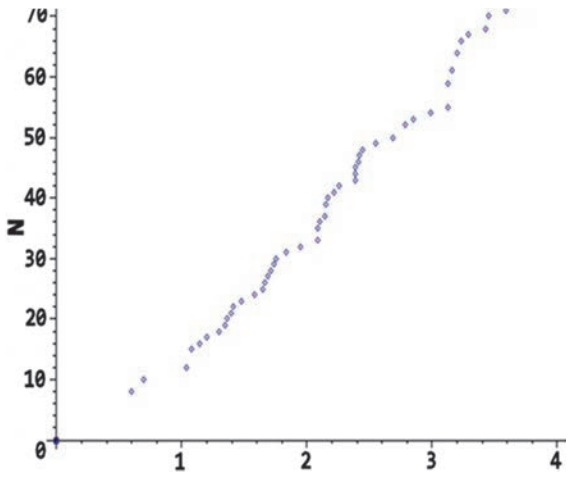


Figure 4-3.3-CAs with 2 Symbole

4.4.4 Applications

A major application of covering arrays is the task of software testing (see the Hartman paper). Any software package will have two natural parameters, the number of inputs and the number of values each input can take. For simplicity, this site only deals with the homogeneous case of when all of the inputs take on the same number of values. This first parameter corresponds to k and the second corresponds to v, as explained above (and below). A third parameter t needs to be chosen and this dictates the “level” of coverage of the input space. Ideally it would be possible to take $t = k$, but that typically results in too many configurations to test. Empirical studies show that for most software packages t need not be more than 6 to find all errors. Once the parameters are fixed, a covering array can be generated. The rows of the array will correspond to tests that need to be performed on the software package and thus need to be converted into a form the program can use. For example, if a system of networked printing was being tested the inputs might be the operating system, the web-browser, the network type, and the time of day. If we take two values for each input, we could modify the above covering array into the test suite below for this system.

OS	Browser	Network	Time
Mac	Opera	wired	morning
Mac	Firefox	wireless	afternoon
Linux	Opera	wireless	afternoon
Linux	Firefox	wired	afternoon
Linux	Firefox	wireless	morning

4.4.5 The Variables of Covering Arrays - t, v, k, and CA(t,k,v)

The question of "What is the size (number of rows) of the smallest covering array" is parameterized by the variables t, v, and k:

- $t - t$ is sometimes called the "strength" of the array. It is a measure of how "fully" we cover the possible rows in the array. For $t = 0$, no rows need to be present and for $t = k$ all rows need to be present (as the number of columns is equal to k). This is because any covering array must satisfy the t-covering property: when any t of the k columns are chosen, all v^t of the possible t-tuples must appear among the rows.

  For example, the following array is not a covering array for $t = 2$, $v = 2$, $v^t = 2^2 = 4$, and $k = 4$, because the second and forth columns do not have the $2 - tuple$ $(1,0)$ among the rows.

```
      K=2     K=4
0     0    0    0
```

```
0    1   1   1
1    1   0   1
1    0   1   1
```

- $v - v$ is the number of possible values that each entry in the array can take on. While the actual symbols chosen are irrelevant, they are typically the numbers between 0 and $v - 1$. v corresponds to the number of settings an input in the software can take on. While typically this varies for each input, these pages only consider the homogeneous case.
- $k - k$ is the number of columns in the array. Thus, the "size" of an array is usually given as its number of rows, as the number of columns is fixed. k corresponds to the number of inputs in the software package that is being tested.
- $CA(t, k, v) - CA(t, k, v)$ is the so-called "covering array number" of the variables t, k, and v. It is the number of rows in the associated covering array. The smallest possible array gives the covering array number denoted $CAN(t, k, v)$, and so these pages present upper-bounds for CAN numbers.

Although most combinatorial testing problems have varying numbers of values per variable, in some cases all variables have the same number of values, and a pre-computed array can be found.

A library of pre-computed covering arrays can be found on Colbourn’s website. The arrays were computed with IPOG-F, a variant of the IPOG algorithm. IPOG and IPOG-F are both available in the ACTS tool, which can be downloaded from this web site. Arrays are available for $t = 2$ to $t = 5$, with 2 to 6 values per variable, and for $t = 6$ with 2 to 5 values per variable.

4.4.6 Testing with Covering Arrays

The following steps should be considered to perform the software testing process using covering arrays:

(1) Define the input variables.

(2) Define the possible values of each variable; many times, the possible values have a continuous value; therefore, these values must be discretized, i.e., divided into classes which represent a range of values.

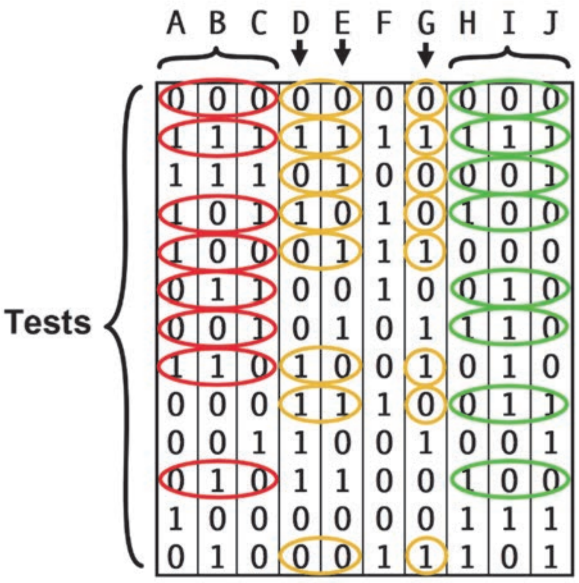
(3) Decide the desired degree of interaction between the system variables.

(4) Build the covering array. Each row of the covering array represents a test case; each column represents a variable involved in the test case, and a value in the column is the particular configuration of the variable. Since each row represents a test case, it is crucial to minimize the number of test cases represented by N, so the testing process requires lower costs and lesser time.

An example is given in ***Figure 4-4***, which shows a 3-way covering array for 10 variables with two values each. The interesting property of this array is that any three columns contain all eight possible values for three binary variables. For example, taking columns F, G, and H, we can see that all eight possible 3-way combinations (000, 001, 010, 011, 100, 101, 110, 111) occur somewhere in the three columns together. In fact, any combination of three columns chosen in any order will also contain all eight possible values. Collectively, therefore, this set of tests will exercise all 3-way combinations of input values in only 13 tests, as compared with 1,024 for exhaustive coverage.

Similar arrays can be generated to cover up to all 6-way combinations. In general, the number of t-way combinatorial tests that will be required is proportional to $v^t \log n$, for n parameters with v possible values each.

Figure 4-5 contrasts these two approaches. With the first approach, we may run the same test set against all 3-way combinations of configuration options, while for the second approach, we would construct a test suite that covers all 3-way combinations of input transaction fields. Of course these approaches could be combined, with the combinatorial tests (approach 2) run against all the configuration combinations (approach 1).


A
B
C
D
E
F
G
H
I
J
Tests

A	B	C	D	E	F	G	H	I	J
0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

Figure 4-4. 3-way covering array

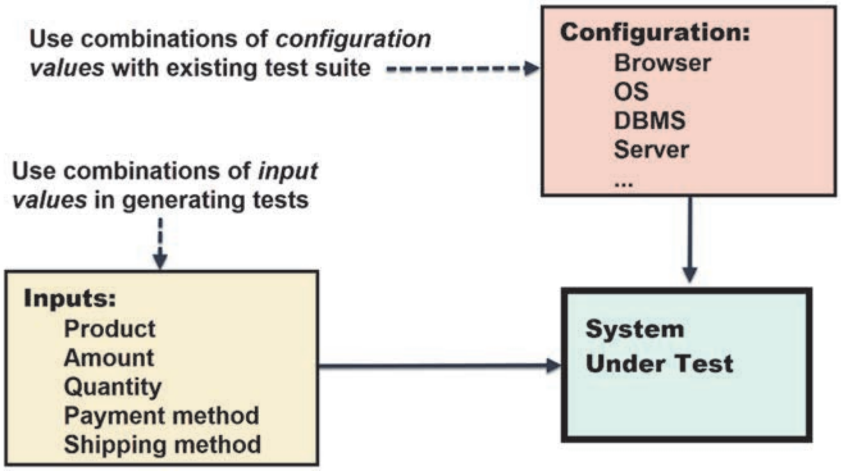

Use combinations of *configuration values* with existing test suite
Configuration:
Browser
OS
DBMS
Server
...
Use combinations of *input values* in generating tests
Inputs:
Product
Amount
Quantity
Payment method
Shipping method
System Under Test

Figure 4-5. Two ways of using combinatorial testing

4.5 The Test Oracle Problem

Even with efficient algorithms to produce covering arrays, the oracle problem remains – testing requires both test data and results that should be expected for each data input. High interaction strength combinatorial testing may require a large number of tests in some cases, although not always. Approaches to solving the oracle problem for combinatorial testing include:

Crash testing: the easiest and least expensive approach is to simply run tests against the **system under test** (SUT) to check whether any unusual combination of input values causes a crash or other easily detectable failure. This is essentially the same procedure used in "fuzz testing", which sends random values against the SUT. This form of combinatorial testing could be regarded as a disciplined form of fuzz testing (Katona, 1973). It should be noted that although pure random testing will generally cover a high percentage of t-way combinations, 100% coverage of combinations requires a random test set much larger than a covering array. For example, all 3-way combinations of 10 parameters with 4 values each can be covered with 151 tests. Purely random generation requires over 900 tests to provide full 3-way coverage.

Embedded assertions: An increasingly popular "light-weight formal methods" technique is to embed assertions within code to ensure proper relationships between data, for example as preconditions, postconditions, or input value checks. Tools such as the **Java Modeling Language** (JML) can be used to introduce very complex assertions, effectively embedding a formal specification within the code. The embedded assertions serve as an executable form of the specification, thus providing an oracle for the testing phase. With embedded assertions, exercising the application with all t-way combinations can provide reasonable assurance that the code works correctly across a very wide range of inputs. This approach has been used successfully for testing smart cards, with embedded JML assertions acting as an oracle for combinatorial tests (Bousquet, Ledru, Maury, Oriat, & Lanet, 2004). Results showed that 80% -90% of errors could be found in this way.

Model based test generation uses a mathematical model of the SUT and a simulator or model checker to generate expected results for each input $[1, 8, 9, 52, 55]$. If a simulator can be used, expected results can be generated directly from the simulation, but model checkers are widely available and can also be used to prove properties such as liveness in parallel processes, in addition to generating tests. Conceptually, a model checker can be viewed as exploring all states of a system model to determine if a property claimed in a specification statement is true. What makes a model checker particularly valuable is that if the claim is false, the model checker not only reports this, but also provides a "counterexample" showing how the claim can be shown false. If the claim is false, the model checker indicates this and provides a trace of parameter input values and states that will prove it is false. In effect this is a complete test case, i.e., a set of parameter values and expected result. It is then simple to map these values into complete test cases in the syntax needed for the system under test. Later chapters develop detailed procedures for applying each of these testing approaches.

4.6 Chapter Summary

1. Empirical data suggest that software failures are caused by the interaction of relatively few parameter values, and that the proportion of failures attributable to t-way interactions declines very rapidly with increase in t. That is, usually single parameter values or a pair of values are the cause of a failure, but increasingly smaller proportions are caused by 3-way, 4-way, and higher order interactions.
2. Because a small number of parameters are involved in failures, we can attain a high degree of assurance by testing all t-way interactions, for an appropriate interaction strength t (2 to 6 usually). The number of t-way tests that will be required is proportional to $v^t \log n$, for n parameters with v values each.
3. Combinatorial methods can be applied to configurations or to input parameters, or in some cases both.
4. As with all other types of testing, the oracle problem must be solved – i.e., for every test input, the expected output must be determined in order to check if the application is producing the

correct result for each set of inputs. A variety of methods are available to solve the oracle problem.

5 SIMPLE APPLICATION PLATFORM EXAMPLE

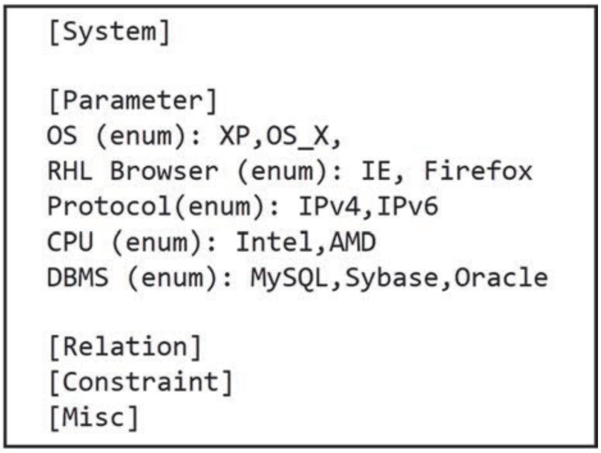
Returning to the simple example introduced in Chapter 4, we illustrate development of test configurations, and compare the size of test suites for various interaction strengths versus testing all possible configurations. For the five configuration parameters, we have $3 \cdot 2 \cdot 2 \cdot 2 \cdot 3 = 72$ configurations. The convention for describing the variables and values in combinatorial testing $v_1^{n_1} v_2^{n_2}$ is where the v_i are number of variable values and n_i are number of occurrences. Thus this configuration is designated $2^3\ 3^2$. Note that at $t = 5$, the number of tests is the same as exhaustive testing for this example, because there are only five parameters. The savings as a percentage of exhaustive testing are good, but not that impressive for this small example. With larger systems the savings can be enormous, as will be seen in the next section.

Table 5-1. Simple example configuration options.

Parameter	Values
Operating system	XP, OS X, RHL
Browser	IE, Firefox
Protocol	IPv4, IPv6
CPU	Intel, AMD
DBMS	MySQL, Sybase, Oracle

We can now generate test configurations using the ACTS tool. For simplicity of presentation we illustrate usage of the command line version of ACTS, but an intuitive GUI version is available that may be more convenient. This tool is summarized in Appendix C and a comprehensive user manual is included with the ACTS download.

The first step in creating test configurations is to specify the parameters and possible values in a file for input to ACTS, as shown in ***Figure 5-1***:



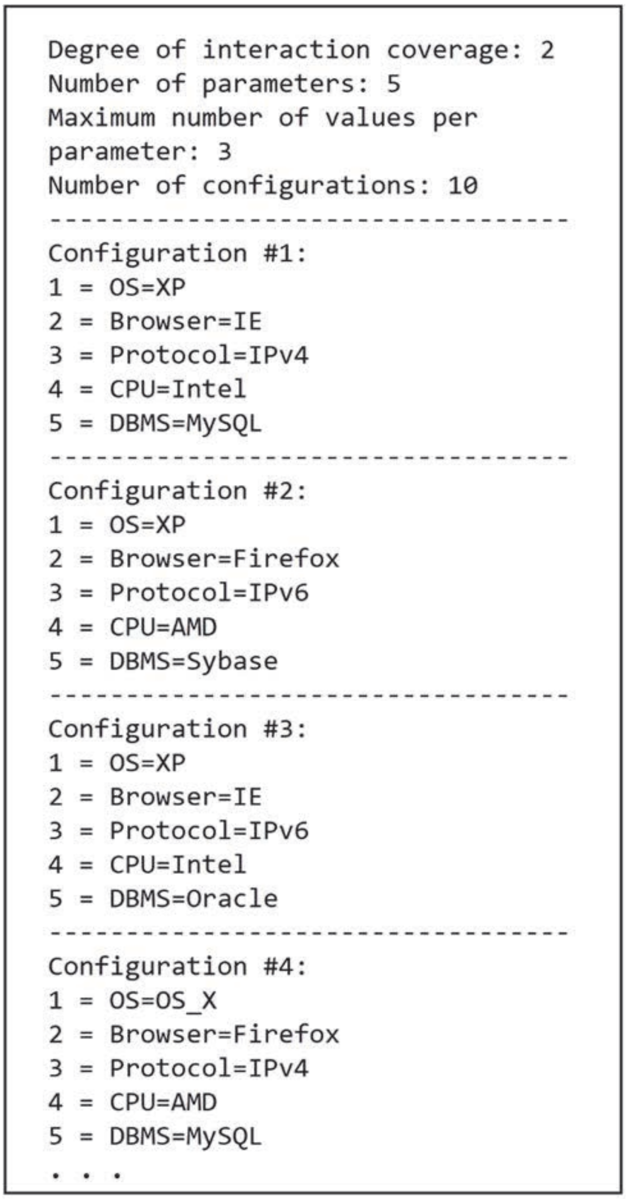
```
[System]

[Parameter]
OS (enum): XP,OS_X,
RHL Browser (enum): IE, Firefox
Protocol(enum): IPv4,IPv6
CPU (enum): Intel,AMD
DBMS (enum): MySQL,Sybase,Oracle

[Relation]
[Constraint]
[Misc]
```

Figure 5-1. Simple example input file for ACTS.

Note that most of the bracketed tags in the input file are optional, and not filled in for this example. The essential part of the file is the [Parameter] specification, in the format `<parameter name> (<type>): <values>`, where one or more values are listed separated by commas. The tool can then be run at the command line:

```
java -Ddoi=2 -jar acts_cmd.jar ActsConsoleManager
in.txt out.txt
```

A variety of options can be specified, but for this example we only use the "degree of interaction" option to specify 2-way, 3-way, etc. coverage. Output can be created in a convenient form shown below, or as a matrix of numbers, comma separated value, or Excel spreadsheet form. If the output will be used by human testers rather than as input for further machine processing, the format in ***Figure 5-2*** is useful:



```
Degree of interaction coverage: 2
Number of parameters: 5
Maximum number of values per
parameter: 3
Number of configurations: 10
----------------------------------
Configuration #1:
1 = OS=XP
2 = Browser=IE
3 = Protocol=IPv4
4 = CPU=Intel
5 = DBMS=MySQL
----------------------------------
Configuration #2:
1 = OS=XP
2 = Browser=Firefox
3 = Protocol=IPv6
4 = CPU=AMD
5 = DBMS=Sybase
----------------------------------
Configuration #3:
1 = OS=XP
2 = Browser=IE
3 = Protocol=IPv6
4 = CPU=Intel
5 = DBMS=Oracle
----------------------------------
Configuration #4:
1 = OS=OS_X
2 = Browser=Firefox
3 = Protocol=IPv4
4 = CPU=AMD
5 = DBMS=MySQL
. . .
```

Figure 5-2. Excerpt of test configuration output covering all 2-way combinations.

The complete test set for 2-way combinations is shown in ***Table 4-1*** in ***Section 4.3***. Only 10 tests are needed. Moving to 3-way or higher interaction strengths requires more tests, as shown in ***Table 5-2***.

Table 5-2. Number of combinatorial tests for a simple example.

t	# Tests	% of Exhaustive
2	10	14
3	18	25
4	36	50
5	72	100

In this example, substantial savings could be realized by testing t-way configurations instead of all possible configurations, although for some applications (such as a small but highly critical module) a full exhaustive test may be warranted. As we will see in the next example, in many cases it is impossible to test all configurations, so we need to develop reasonable alternatives.

5.1 Smart Phone Application Example

Smart phones have become enormously popular because they combine communication capability with powerful graphical displays and processing capability. Literally tens of thousands of smart phone applications, or 'apps', are developed annually. Among the platforms for smart phone apps is the Android, which includes an open-source development environment and specialized operating system. Android units contain a large number of configuration options that control the behavior of the device. Android apps must operate across a variety of hardware and software platforms, since not all products support the same options. For example, some smart phones may have a physical keyboard and others may present a soft keyboard using the touch sensitive screen. Keyboards may also be either only numeric with a few special keys, or a full typewriter keyboard. Depending on the state of the app and user choices, the keyboard may be visible or hidden. Ensuring that a particular app works across the enormous number of options is a significant challenge for developers. The extensive set of options makes it intractable to test all possible configurations, so combinatorial testing is a practical alternative.

Table 5-3 shows a resource configuration file for Android apps. A total of 35 options may be set. Our task is to develop a set of test configurations that allow testing across all 4-way combinations of these options. The first step is to determine the set of parameters and possible values for each that will be tested. Although the options are listed individually to allow a specific integer value to be associated with each, they clearly represent sets of option values with mutually exclusive choices. For example, "Keyboard Hidden" may be "yes", "no", or "undefined". These values will be the possible settings for parameter names that we will use in generating a covering array. ***Table 5-4*** shows the parameter names and number of possible values that we will use for input to the covering array generator. For a complete specification of these parameters, see:

http://developer.android.com/reference/android/content/res/Configuration.html

Table 5-3. Android resource configuration file.

```
     int       HARDKEYBOARDHIDDEN_NO;
     int       HARDKEYBOARDHIDDEN_UNDEFINED;
     int       HARDKEYBOARDHIDDEN_YES;
     int       KEYBOARDHIDDEN_NO;
     int       KEYBOARDHIDDEN_UNDEFINED;
     int       KEYBOARDHIDDEN_YES;
     int       KEYBOARD_12KEY;
     int       KEYBOARD_NOKEYS;
     int       KEYBOARD_QWERTY;
     int       KEYBOARD_UNDEFINED;
     int       NAVIGATIONHIDDEN_NO;
     int       NAVIGATIONHIDDEN_UNDEFINED;
     int       NAVIGATIONHIDDEN_YES;
     int       NAVIGATION_DPAD;
     int       NAVIGATION_NONAV;
     int       NAVIGATION_TRACKBALL;
     int       NAVIGATION_UNDEFINED;
     int       NAVIGATION_WHEEL;
     int       ORIENTATION_LANDSCAPE;
     int       ORIENTATION_PORTRAIT;
     int       ORIENTATION_SQUARE;
     int       ORIENTATION_UNDEFINED;
     int       SCREENLAYOUT_LONG_MASK;
```

```
int        SCREENLAYOUT_LONG_NO;
int        SCREENLAYOUT_LONG_UNDEFINED;
int        SCREENLAYOUT_LONG_YES;
int        SCREENLAYOUT_SIZE_LARGE;
int        SCREENLAYOUT_SIZE_MASK;
int        SCREENLAYOUT_SIZE_NORMAL;
int        SCREENLAYOUT_SIZE_SMALL;
int        SCREENLAYOUT_SIZE_UNDEFINED;
int        TOUCHSCREEN_FINGER;
int        TOUCHSCREEN_NOTOUCH;
int        TOUCHSCREEN_STYLUS;
int        TOUCHSCREEN_UNDEFINED;
```

Table 5-4. Android configuration options.

Parameter Name	Values	# Values
HARDKEYBOARDHIDDEN	NO, UNDEFINED, YES	3
KEYBOARDHIDDEN	NO, UNDEFINED, YES	3
KEYBOARD	12KEY, NOKEYS, QW ERTY, UNDEFINED	4
NAVIGATIONHIDDEN	NO, UNDEFINED, YES	3
NAVIGATION	DPAD, NONAV, TRACKBALL, UNDEFINED, WHEEL	5
ORIENTATION	LANDSCAPE, PORTRAIT, SQUARE, UNDEFINED	4
SCREENLAYOUT_LONG	MASK, NO, UNDEFINED, YES	4
SCREENLAYOUT_SIZE	LARGE, MASK, NORMAL, SMALL, UNDEFINED	5
TOUCHSCREEN	FINGER, NOTOUCH, STYLUS, UNDEFINED	4

Using ***Table 5-4***, we can now calculate the total number of configurations:

$$3 \cdot 3 \cdot 4 \cdot 3 \cdot 5 \cdot 4 \cdot 4 \cdot 5 \cdot 4 = 172{,}800 \text{ configurations}$$

(i.e., a $3^3 4^4 5^2$ system). Like many applications, thorough testing will require some human intervention to run tests and verify results, and a test suite will typically include many tests. If each test suite can be run in 15 minutes, it will take roughly 24 staff-years to complete testing for an app. With salary and benefit costs for each tester of $150,000, the cost of testing an app will be more than $3 million, making it virtually impossible to return a profit for most apps. How can we provide effective testing for apps at a reasonable cost?

Using the covering array generator, we can produce tests that cover *t*-way combinations of values. ***Table 5-5*** shows the number of tests required at several levels of *t*. For many applications, 2-way or 3-way testing may be appropriate, and either of these will require less than 1% of the time required to cover all possible test configurations.

Table 5-5. Number of combinatorial tests for Android example.

t	# Tests	% of Exhaustive
2	29	0.02
3	137	0.08
4	625	0.4
5	2532	1.5
6	9168	5.3

5.2 Cost and Practical Considerations

5.2.1 Invalid Combinations and Constraints

The system described in ***Section 3.2*** illustrates a common situation in all types of testing: some combinations cannot be tested because they don't exist for the systems under test. In this case, if the operating system is either OS X or Linux, Internet Explorer is not available as a browser. Note that we cannot simply delete tests with these untestable combinations, because that would result in losing other combinations that are essential to test but are not covered by other tests. For example, deleting tests 5 and 7 in ***Section 4.3*** would mean that we would also lose the test for Linux with the IPv6 protocol.

One way around this problem is to delete tests and supplement the test suite with manually constructed test configurations to cover the deleted combinations, but covering array tools offer a better solution. With ACTS we can specify constraints, which tell the tool not to include specified combinations in the generated test configurations. ACTS supports a set of commonly used logic and arithmetic operators to specify constraints. In this case, the following constraint can be used to ensure that invalid combinations are not generated:

```
(OS != “XP” => Browser = “Firefox”)
```

The covering array tool will then generate a set of test configurations that does not include the invalid combinations but does cover all those that are essential. The revised test configuration array is shown in

Figure 5-3 below. Parameter values that have changed from the original configurations are underlined. Note that adding the constraint also resulted in reducing the number of test configurations by one. This will not always be the case, depending on the constraints used, but it illustrates how constraints can help reduce the problem. Even if particular combinations are testable, the test team may consider some combinations unnecessary, and constraints could be used to prevent these combinations, possibly reducing the number of test configurations.

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	Firefox	IPv4	Intel	Sybase
6	OS X	Firefox	IPv6	AMD	Oracle
7	RHL	Firefox	IPv6	Intel	MySQL
8	RHL	Firefox	IPv4	Intel	Oracle
9	XP	IE	IPv4	AMD	Sybase

Figure 5-3. Test configurations for simple example with constraint.

5.2.2 Cost Factors

Using combinatorial methods to design test configurations is probably the most widely used combinatorial approach because it is quick and easy to do and typically delivers significant improvements to testing. Combinatorial testing for input parameters can provide better test coverage at lower cost than conventional tests and can be extended to high strength coverage to provide much better assurance.

5.3 Chapter Summary

1. Configuration testing is probably the most commonly used application of combinatorial methods in software testing. Whenever an application has roughly five or more configurable attributes, a covering array is likely to make testing more efficient. Configurable attributes usually have a small number of possible values each, which is an ideal

situation for combinatorial methods. Because the number of t-way tests is proportional to $v^t \log n$, for n parameters with v values each, unless configurable attributes have more than 8 or 10 possible values each, the number of tests generated will probably be reasonable. The real-world testing problem introduced in ***Section 3.2*** is a fairly typical size, where 4-way interactions can be tested with a few hundred tests.

2. Because many systems have certain configurations that may not be of interest (such as Internet Explorer browser on a Linux system), constraints are an important consideration in any type of testing. With combinatorial methods, it is important that the covering array generator allows for the inclusion of constraints so that all relevant interactions are tested, and important information is not lost because a test contains an impossible combination.

6 INPUT PARAMETER TESTING

As noted in the introduction, the key advantage of combinatorial testing derives from the fact that all, or nearly all, software failures appear to involve interactions of only a few parameters. Using combinatorial testing to select configurations can make testing more efficient, but it can be even more effective when used to select input parameter values. Testers traditionally develop scenarios of how an application will be used, then select inputs that will exercise each of the application features using representative values, normally supplemented with extreme values to test performance and reliability. The problem with this often-ad hoc approach is that unusual combinations will usually be missed, so a system may pass all tests and work well under normal circumstances, but eventually encounter a combination of inputs that it fails to process correctly.

By testing all t-way combinations, for some specified level of t, combinatorial testing can help to avoid this type of situation. In this chapter we work through a small example to illustrate the use of these methods.

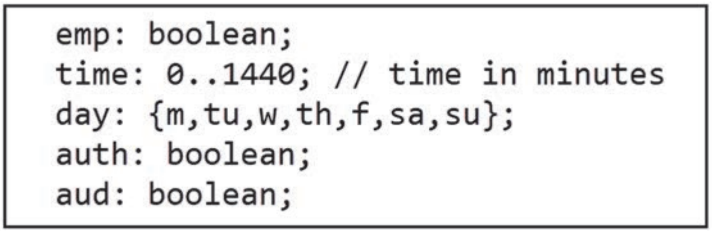
6.1 Example Access Control Module

The system under test is an access control module that implements the following policy:

Access is allowed if and only if:

- the subject is an employee AND current time is between 9 am and 5 pm AND it is not a weekend
- OR subject is an employee with a special authorization code
- OR subject is an auditor AND the time is between 9 am and 5 pm (not constrained to weekdays).

The input parameters for this module are shown in ***Figure 6-1:***



```
emp: boolean;
time: 0..1440; // time in minutes
day: {m,tu,w,th,f,sa,su};
auth: boolean;
aud: boolean;
```

Figure 6-1. Access control module input parameters.

Our task is to develop a covering array of tests for these inputs. The first step will be to develop a table of parameters and possible values, similar to that in ***Section 5.1*** in the previous chapter. The only difference is that in this case we are dealing with input parameters rather than configuration options. For the most part, the task is simple: we just take the values directly from the specifications or code, as shown in ***Figure 6-2***. Several parameters are Boolean, and we will use 0 and 1 for false and true values respectively. For day of the week, there are only seven values, so these can all be used. However, the hour of the day presents a problem. Recall that the number of tests generated for n parameters is proportional to v^t, where v is the number of values and t is the interaction level (2-way to 6-way). For all Boolean values and 4-way testing, therefore, the number of tests will be a multiple of 24 . But consider what happens with a large number of possible values, such as 24 hours. The number of tests will be proportional to 244 = 331,736. For this example, time is given in minutes, which would obviously be completely intractable. Therefore, we must select representative values for the hour parameter. This problem occurs in all types of testing, not just with combinatorial methods, and good methods have been developed to deal with it. Most testers are already familiar with two of these: equivalence partitioning and boundary value analysis. Additional background on these methods can be found in software testing texts such as Ammann and Offutt (2008), Beizer (2003), Copeland (2004), Lyu (1996, p. 1), and Meyer (Meyer, 1997).

Parameter	Values
emp	0,1
time	??
day	m,tu,w,th,f,sa,su
auth	0, 1
aud	0, 1

Figure 6-2. Parameters and values for access control example.

Both of these intuitively obvious methods will produce a smaller set of values that should be adequate for testing purposes, by dividing the possible values into partitions that are meaningful for the program being tested. One value is selected for each partition. The objective is to partition the input space such that any value selected from the partition will affect the program under test in the same way as any other value in the partition. Thus, ideally if a test case contains a parameter x which has value y, replacing y with any other value from the partition will not affect the test case result. This ideal may not always be achieved in practice.

How should the partitions be determined? One obvious, but not necessarily good, approach is to simply select values from various points on the range of a variable. For example, if capacity can range from 0 to 20,000, it might seem sensible to select 0, 10,000, and 20,000 as possible values. But this approach is likely to miss important cases that depend on the specific requirements of the system under test. Some judgment is involved, but partitions are usually best determined from the specification. In this example, 9 am and 5 pm are significant, so 0540 (9 hours past midnight) and 1020 (17 hours past midnight) determine the appropriate partitions:

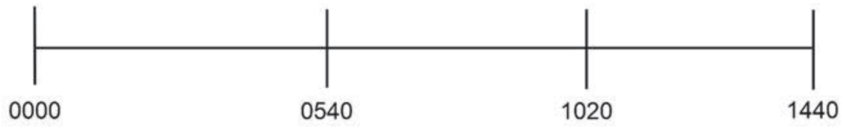


Figure 6-3. Parameters and values for access control example.

Ideally, the program should behave the same for any times within the partitions; it should not matter whether the time is 4:00 am or 7:03 am, for example, because the specification treats both of these times the same. Similarly, it should not matter which time between the hours of 9 am and 5 pm is chosen; the program should behave the same for 10:20 am and 2:33 pm. One common strategy, boundary value analysis, is to select test values at each boundary and at the smallest possible unit on either side of the boundary, for three values per boundary. The intuition, backed by empirical research, is that errors are more likely at boundary conditions because errors in programming may be made at these points. For example, if the requirements for automated teller machine software say that a withdrawal should not be allowed to exceed $300, a programming error such as the following could occur:

```
if (amount > 0 && amount < 300) {
      //process withdrawal } else {
// error message
}
```

Here, the second condition should have been “amount <= 300”, so a test case that includes the value amount = 300 can detect the error, but a test with amount = 305 would not.

It is generally also desirable to test the extremes of ranges. One possible selection of values for the time parameter would then be: 0000, 0539, 0540, 0541, 1019, 1020, 1021, and 1440. More values would be better, but the tester may believe that this is the most effective set for the available time budget. With this selection, the total number of combinations is 2 ·8 ·7 ·2 · 2 = 448.

Generating covering arrays for $t = 2$ through 6, as detailed in ***Section 6.1*** results in the following number of tests:

# Tests	
2	56
3	112
4	224

Figure 6-4. Number of tests for access control example.

6.2 Real-world Systems

As with the previous example, the advantage over exhaustive testing is not large, because of the small number of parameters. With larger problems, the advantages of combinatorial testing can be spectacular. For example, consider the problem of testing the software that processes switch settings for the panel shown in ***Figure 6-5***. There are 34 switches, which can each be either on or off, for a total of $2^{34} = 1.7\ x\ 10^{10}$ possible settings. We clearly cannot test 17 billion possible settings, but all 3way interactions can be tested with only 33 tests, and all 4-way interactions with only 85. This may seem surprising at first, but it results from the fact that every test of 34 Parameters contain $\binom{34}{3} = 5{,}984$ three-way and $\binom{34}{4} = 46{,}376$ four-way combinations.

Figure 6-5. Panel with 34 switches.

6.3 Cost and Practical Considerations

Combinatorial methods can be highly effective and reduce the cost of testing substantially. For example, Justin Hunter has applied these methods to a wide variety of test problems and consistently found both lower cost and more rapid error detection (Kuhn, Kacker, & J. Hunter, Combinatorial Software Testing, 2009). But as with most aspects of engineering, tradeoffs must be considered. Among the most important is the question of when to stop testing, balancing the cost of testing against the risk of failing to discover additional failures. An extensive body of research has been devoted to this topic, and sophisticated models are available for determining when the cost of further testing will exceed the expected benefits [(Boehm, 1981), (Lyu, 1996)]. Existing models for when to stop testing can be applied to the combinatorial test approach also, but there is an additional consideration: What is the appropriate interaction strength to use in this type of testing?

To address these questions consider the number of tests at different interaction strengths for an avionics software example (Kuhn & Okun, Pseudo-exhaustive Testing for Software, 2006) shown in ***Figure 6-6***. While the number of tests will be different (probably much smaller than in ***Figure 6-6***) depending on the system under test, the magnitude of difference between levels of t will be similar to ***Figure 6-6***, because the number of tests grows with v^t, for parameters with v values. That is, the number of tests grows with the exponent t, so we want to use the smallest interaction strength that is appropriate for the problem. Intuitively, it seems that if no failures are detected by t-way tests, then it may be reasonable to conduct additional testing only for $t+1$ interactions, but no greater if no additional failures are found at $t+1$. In the empirical studies of software failures, the number of failures detected at $t>2$ decreased monotonically with t, so this heuristic seems to make sense: *start testing using 2-way (pairwise) combinations, continue increasing the interaction strength* t *until no errors are detected by the* t*-way tests, then (optionally) try* $t+1$ *and ensure that no additional errors are detected*. As with other aspects of software development, this guideline is also dependent on resources, time constraints, and cost-benefit considerations.

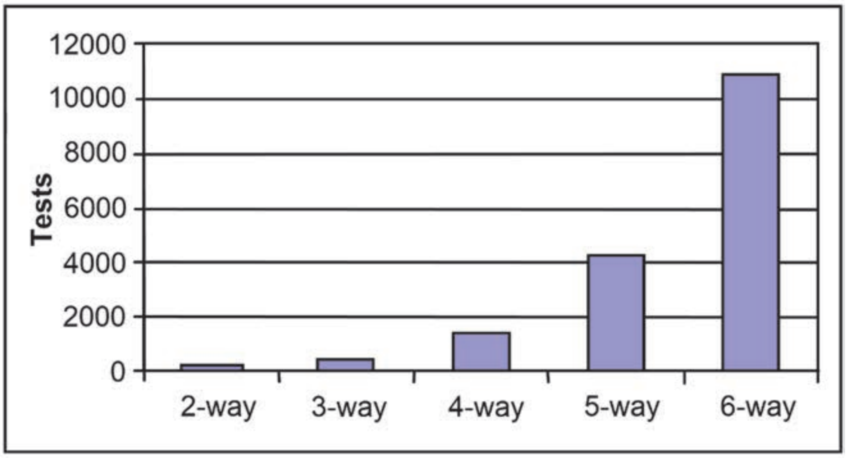


Figure 6-6. Number of tests for avionics example.

When applying combinatorial methods to input parameters, the key cost factors are the number of values per parameter, the interaction strength, and the number of parameters. As shown above, the number of tests increases rapidly as the value of t is increased, but the rate of increase depends on the number of values per parameter. Binary variables, with only two values each, result in far fewer tests than parameters with many values each. As a practical matter, when partitioning the input space (section 6.1[1]), it is best to keep the number of values per parameter below 8 or 10 if possible.

Because the number of tests increases only logarithmically with the number of parameters, test set size for a large problem may be only somewhat larger than for a much smaller problem. For example, if a project uses combinatorial testing for a system that has 20 parameters and generates several hundred tests, a much larger system with 40 to 50 parameters may only require a few dozen more tests. Combinatorial methods may generate the best cost benefit ratio for large systems.

1

6.4 Chapter Summary

1. The key advantage of combinatorial testing derives from the fact that all, or nearly all, software failures appear to involve interactions of only a few parameters. Generating a covering array of input parameter values allows us to test all of these interactions, up to a level of 5-way or 6-way combinations, depending on resources.
2. Practical testing often requires abstracting the possible values of a variable into a small set of equivalence classes. For example, if a variable is a 32-bit integer, it is clearly not possible to test the full range of values in $+/-2^{31}$. This problem is not unique to combinatorial testing but occurs in most test methodologies. Simple heuristics and engineering judgment are required to determine the appropriate portioning of values into equivalence classes, but once this is accomplished it is possible to generate covering arrays of a few hundred to a few thousand tests for many applications. The thoroughness of coverage will depend on resources and criticality of the application.

7 USING ACTS FOR COMBINATORIAL TEST DESIGNS

ACTS is a test generation tool for constructing t-way combinatorial test sets. Currently, it supports t-way test set generation with t ranging from 1 to 6. Combinatorial testing has been shown to be very effective in detecting faults that are caused by unexpected interactions between different factors. The tool provides three interfaces, including Graphic User Interface, Command Line Interface (CLI) and Application Programming Interface (API).

Note that ACTS is released in two versions, *Basic Version* and *Advanced Version*. Features that are marked with * are only available in the *Advanced Version*.

7.1 Core Features

7.1.1 T-Way Test Set Generation

This is the core feature of ACTS. A system 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 (out of all the parameters) of a system, every combination of values of these t parameters is covered in at least one test in the test set.

Currently, ACTS supports t-way test set generation for $1 \leq t \leq 6$. Empirical studies show that t being up to 6 is sufficient for most practical applications. A special form of 1-way testing, called base-choice testing, is implemented in ACTS. Base-choice testing requires that every parameter value be covered at least once and in a test in which all the other values are base choices. Each parameter has one or more values designated as base choices. Informally, base choices are "more important" values, e.g., default values, or values that are used most often in operation.

In the full version of ACTS, several test generation algorithms are implemented in ACTS. These algorithms include IPOG, IPOG-D, IPOG-F, IPOG-F2. In general, IPOG, IPOG-F, and IPOG-F2 work best for systems of moderate size (less than 20 parameters and 10 values per parameter on average), while IPOG-D is preferred for larger systems. Note that

IPOG-D and IPOG-F2 do not support mixed strength, constraint, or negative testing. IPOG is the only algorithm available in the Free version of ACTS.

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. In the *extend* mode, an existing test set can be a test set that is generated by ACTS but is incomplete because of some newly added parameters and values, or because of a test set that is supplied by the user and imported into ACTS. Extending an existing test set can save earlier effort that has already been spent in the testing process.

7.1.2 Mixed Strength*

In the Advanced Version of ACTS, this feature allows different parameter groups to be created and covered with different strengths. For example, consider a system consisting of 10 parameters, P1, P2, …, and P10. The first relation can be created that consists of all the parameters with strength 2. Then, additional relations can be created if some parameters are believed to have a higher degree of interaction, based on the user's domain knowledge. For instance, a relation could be created that consists of P2, P4, P5, P7, P8 with strength 3 if the four parameters are closely related to each other, and their 3-way interactions could trigger certain software faults. 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. Mixed strength is only supported by the IPOG and IPOG-F algorithm.

*7.2 Constraint Support**

Some combinations are not valid from the domain semantics and must be excluded from the resulting test set. For example, when we want to make sure a web application can run in different Internet browsers and on different Operating Systems, the configuration of IE on Mac OS is not a valid combination. A test that contains an invalid combination will be rejected by the system (if adequate input validation is performed) or may cause the system to fail. In either case, the test will not be executed properly, which may compromise test coverage, if some (valid) combinations are only covered by this test.

In the Advanced Version of ACTS allows the user to specify constraints that combinations must satisfy to be valid. The specified constraints will be considered during test generation so that the resulting test set will cover, and only cover, combinations that satisfy these constraints. Currently, constraint support is only available for the IPOG and IPOG-F algorithm. Constraint support for other algorithms will be added in a future release.

*7.3 Negative Testing**

Negative testing, which is also referred to as robustness testing, is used to test whether a system handles invalid inputs correctly. ACTS allows the user to designate some values of a parameter as invalid values. During test generation, ACTS ensures that each test contains at most one invalid value. This is to avoid potential masking effect between invalid values. In addition, ACTS ensures that every invalid value is combined with every $(t-1)$-way combination of valid values. Currently, t-way negative testing is only supported in the IPOG and IPOG-F algorithm. Note that, due to the characteristic of base-choice testing, it can work on both valid and invalid values to generate a special form of 1-way positive and negative tests.

7.4 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. A test set to be verified can be a test set generated by ACTS or a test set supplied by the user (and then imported into ACTS).

7.5 Command Line Interface

There is a jar file for both command line mode and GUI mode. In this section, we assume that the jar file is named **acts.jar**. (The actual jar files names in the release package may be different.) The command line mode can be executed using the following command[1]:

```
java <options> -jar acts.jar <input_filename>
[output_filename]
```

Alternatively, the command line mode can also be executed by using classpath:

```
java <options> -cp acts.jar
edu.uta.cse.fireeye.console.ActsConsoleManager
<input_filename> [output_filename]
```

The various options are shown below, where options marked with * are only available in the Advanced Version:

-**Dalgo**=ipog|ipog_d|ipof|ipof2|basechoice|null

- ipog – use algorithm IPO (default)
- ipog_d – use algorithm IPO + Binary Construction (for large systems)*
- ipof - use the IPOF algorithm*
- ipof2 - use the IPOF2 algorithm*
- basechoice - use the Base Choice algorithm*
- null - use to check coverage only (no test generation)

-**Ddoi**=<int>

- specify the degree of interactions to be covered. Default value is 2. Use -1 for mixed strength.*

-**Doutput**=numeric|nist|csv|excel

- numeric - output test set in numeric format
- nist - output test set in NIST format (default)

- csv - output test set in CSV format
- excel - output test set in EXCEL format

-**Dmode**=scratch|extend

- scratch - generate tests from scratch (default)
- extend - extend from an existing test set

-**Dchandler**=no|solver|forbiddentuples*

- no - ignore all constraints
- solver - handle constraints using CSP solver
- forbiddentuples - handle constraints using minimum forbidden tuples (default)

-**Dcheck**=on|off

- on - verify coverage after test generation
- off - do not verify coverage (default)

-**Dprogress**=on|off

- on - display progress information
- off - do not display progress information (default)

-**Ddebug**=on|off

- on - display debug info
- off - do not display debug info (default)

The above usage information can be displayed using the following command:

```
java -jar acts.jar -h
```

In the command line, `<input_file>` contains the configuration information of the system to be tested. There are two supported formats of input file: XML and TXT. Configuration file in these two formats can be formally created or modified via GUI. If the user wants to manually create a file for CLI execution, I suggest using the TXT format, instead of the XML format. The TXT format of a configuration file is illustrated using the following example, where the [Relation] and [Constraint] sections are allowed only in the Advanced Version:

```
[System]
-- specify system name
Name: TCAS
[Parameter]
-- general syntax is parameter_name (type): value1,
   value2, ... Cur_Vertical_Sep (int) : 299, 300, 601
High_Confidence (boolean): TRUE, FALSE
   Two_of_Three_Reports_Valid (boolean) : TRUE, FALSE
   Own_Tracked_Alt (int) : 1, 2
Other_Tracked_Alt (int) : 1, 2
Own_Tracked_Alt_Rate (int) : 600, 601
Alt_Layer_Value (int) : 0, 1, 2, 3
Up_Separation (int) : 0, 399, 400, 499, 500, 639, 640,
   739, 740, 840
Down_Separation (int) : 0, 399, 400, 499, 500, 639,
   640, 739, 740, 840
Other_RAC (enum): NO_INTENT, DO_NOT_CLIMB,
   DO_NOT_DESCEND
Other_Capability (enum): TCAS_TA, OTHER
Climb_Inhibit (boolean): TRUE, FALSE
[Relation]
-- this section is optional
-- general format Rx : (p1, p2, ..., pk, Strength)
R1 : (Cur_Vertical_Sep, Up_Separation,
   Down_Separation, 3)
[Constraint]
-- this section is also optional
Cur_Vertical_Sep != 299 => Other_Capability != "OTHER"
Climb_Inhibit = true => Up_Separation > 399
[Test Set]
-- set existing test set for extend mode. this is also
```

```
   optional
-- * represents don't-care value
Cur_Vertical_Sep, Other_Tracked_Alt, Alt_Layer_Value,
   Climb_Inhibit 299,1,0,false
```

Currently, three parameter types are supported: **enum**, **bool** (Boolean), and **int** (integer). Note that types **bool** and **enum** are **case insensitive**. Also, type int supports **negative values**. Lines beginning with -- represents comments that exist only to improve the readability of the configuration file.

In order to support negative testing and base-choice testing (since ACTS 3.0), the user may add invalid parameter values and may designate some values as base choices, as shown below:

```
… [Parameter]
….
Cur_Vertical_Sep (int) : [299], [300], 601 ; -1, -2
High_Confidence (boolean) : [TRUE], FALSE
...
Other_Capability (enum) : [TCAS_TA], OTHER ; InvalidStr
…
```

As shown above, invalid values are specified after valid values with a semicolon separating them. Note that each parameter must have at least one valid value, but it may or may not have any invalid values. A Boolean parameter cannot have any invalid value. Square bracket is used to indicate that a value is a base-choice value. A parameter can have one or more base-choice values. Base-choice values must be valid values.

The default heap size for the Java Virtual Machine may not be adequate for large configurations. The user is recommended to change the default heap size, if necessary, using the following command:

```
java -Xms <initial heap size> -Xmx <max heap
size> <options> -jar acts.jar <input_file>
[output_file]
```

The following command can be used to save the progress information displayed on the console:

```
java <options> -Dprogress=on -jar acts.jar
<input_file> [output_file] > progress.log
```

7.6 GUI Interface

There are two ways to launch the GUI. One way is to double-click the jar file, which is executable. The other way is to execute the jar file on the command prompt as follows:

```
java -jar acts.jar
```

The following command can be used to change the default heap size for java virtual machine, if necessary:

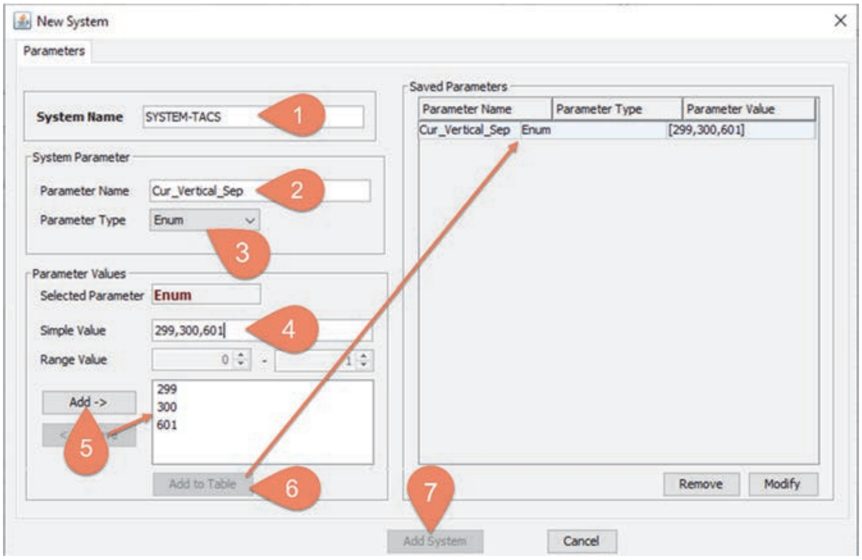
```
java -Xms <initial heap size> -Xmx <max heap size>
<options> -jar acts.jar
```

Figure 7-3 and ***Figure 7-5*** show the general layout of the ACTS GUI. The System View component is a tree structure that shows the configurations of the systems that are currently open in the GUI. In the tree structure, each system is shown as a three-level hierarchy. That is, each system (top level) consists of a set of parameters (second level), each of which has a set of values (leaf level). If a system has relations and constraints, they will be shown in the same level as parameters. Note that base-choice values are shown in bold, and invalid values are shown in red color.

Referring to ***Figure 7-1***:

1. Enter System Name = SYSTEM-TACS
2. Enter a Parameter Name = Cur_Vertical_Sep
3. Select the parameter type = Enum
4. Enter Simple Value (comma separated) = 299,300,601
5. Select the Add-> button. The number from the previous step will populate the window.

6. Select the Add to Table button. The parameter just defined will appear in the Saved Parameters window. Repeat steps 2 through 6 for each parameter.
7. After all parameters are entered, select the Add Stem button to create the new system.



System Name: SYSTEM-TACS
Parameter Name: Cur_Vertical_Sep
Parameter Type: Enum
Selected Parameter: Enum
Simple Value: 299,300,601
Range Value: 0 - 1

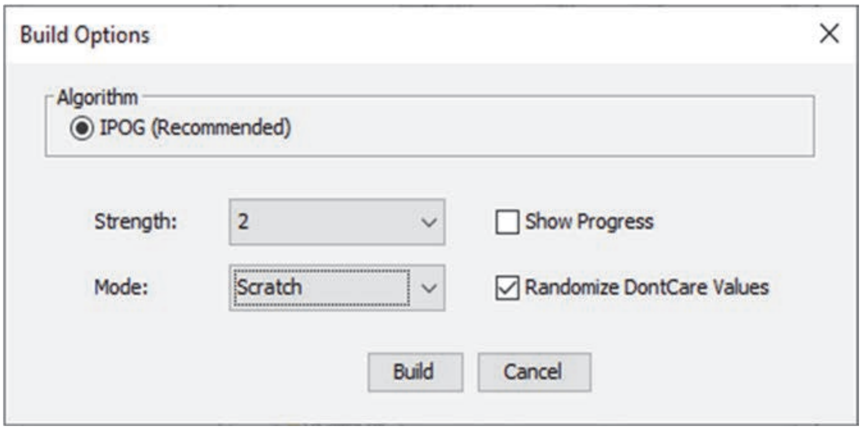
299
300
601

Parameter Name	Parameter Type	Parameter Value
Cur_Vertical_Sep	Enum	[299,300,601]

Figure 7-1. ACTS New System definition window

7.7 Build Test Set

To build a test set for a system that is currently open, select the system in the System View, and then select menu ***Operations -> Build***. The latter selection brings up the ***Build Options*** window, as shown in ***Figure 7-2***, which allows the following options to be specified for the build operation. Note that some options are not supported by some algorithms. Also, note that the Build Options window is simplified in the Basic Version as it only supports the IPOG algorithm without constraints and relations.



Algorithm

[● IPOG (Recommended)]

Strength: 2

Mode: Scratch

☐ Show Progress

☑ Randomize DontCare Values

Figure 7-2. Build Options Window

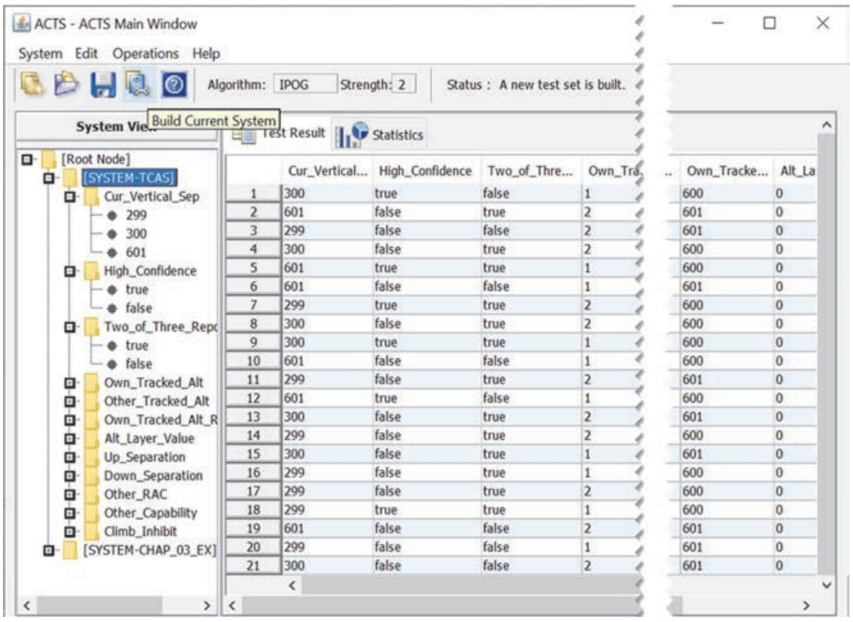
- **Algorithm**: This option decides which algorithm to be used for test generation. By default, the IPOG algorithm is selected and is the only option in the Basic version of ACTS. As mentioned in Section 1.1, IPOG, IPOG-F, IPOG-F2 and work best for systems of moderate size, while IPOG-D is preferred for larger systems. Negative testing is only supported in IPOG and IPOG-F. The IPOG algorithm is recommended for most applications.
- **Randomize Don't Care Values**: If this option is checked, then all the don't care values (*) in the resulting test set will be replaced with random values which don't violate constraints. The coverage will not be affected if we replace a don't care value with a real value.
- **Ignore Constraints**: If this option is checked, all constraints will be ignored. For the generation algorithms that don't support constraints, i.e., IPOG-D and IPOG-F2, this option will be automatically checked.
- **Strength**: This option specifies the strength of the test set. Currently, ACTS supports a strength value ranging from 1 to 6. If the strength is set to a number between 1 and 6, only the specified strength will be used for test generation. If the strength is set to "Mixed", the relations specified in the system configuration (Section 7.1, Relations tab) will be used. For the Base Choice algorithm the strength will be set to 1.
- **Mode**: This option can be Scratch or Extend. The former specifies that a test set should be built from scratch; the latter specifies that a test set should be built by extending an existing test set (shown in

the Test Result tab). Recall that the current test set in the system may not be complete as the system configuration may have changed after the last build or the test set may be imported from outside.

- **Constraint Handling**: This item is not available with the Basic version of ACTS. With the full program, we can select the options Forbidden Tuples or CSP Solver. The former specifies that validity check be handled using minimum forbidden tuples, which are generated from constraints. Forbidden Tuples is set to be the default option. The latter specifies that validity check be handled using a CSP solver. In general, Forbidden Tuples is faster than CSP Solver when the constraints are not complex, especially when constraints only involve a small number of parameters. Both options will produce the same test set. The user may switch between Forbidden Tuples and CSP Solver for Constraint Handling when one takes too long to complete.
- **Progress**: If this option is turned on, progress information will be displayed on the console. Note that in order to obtain the console, the GUI must be started from a command prompt instead of double-clicking the executable jar file.

After the build operation is completed, the resulting test set will be displayed in the Test Result tab of the Main window. Note that negative tests, if exist, are placed after positive tests.

Right to the System View is a tabbed pane consisting of two tabs, namely, Test Result, which is shown in ***Figure 7-3***, and Statistics, which is shown in ***Figure 7-4***. The Test Result shows a test set of the currently selected system, where each row represents a test, and each column represents a parameter. Output parameters are also displayed as columns. The Statistics tab displays some statistical information about the test set. Selecting the Graph button (orange arrow) produces the graph that plots the growth rate of the test coverage with respect to the tests in the test set displayed in the Test Result tab (see ***Figure 7-5***). Drawing the graph may involve expensive computations, and thus the graph is shown only on demand, i.e. when the Graph button is clicked.

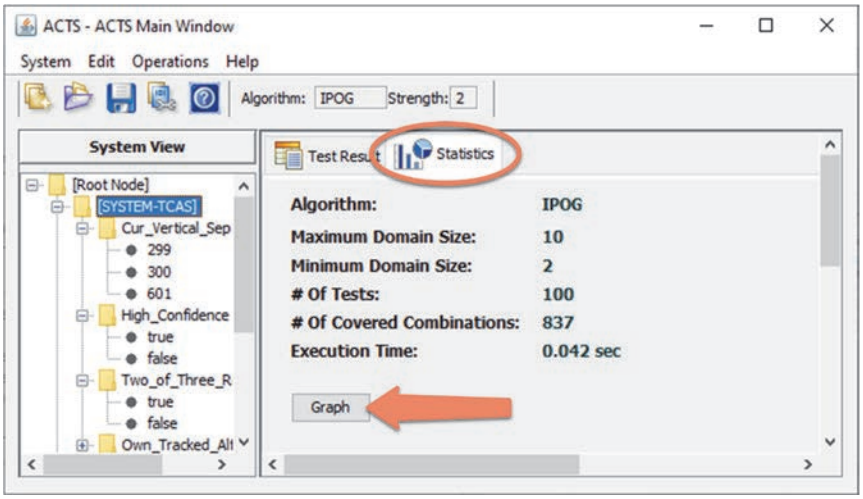


Algorithm: IPOG Strength: 2 Status : A new test set is built.

[Root Node]
- [SYSTEM-TCAS]
  - Cur_Vertical_Sep
    - 299
    - 300
    - 601
  - High_Confidence
    - true
    - false
  - Two_of_Three_Repo
    - true
    - false
  - Own_Tracked_Alt
  - Other_Tracked_Alt
  - Own_Tracked_Alt_R
  - Alt_Layer_Value
  - Up_Separation
  - Down_Separation
  - Other_RAC
  - Other_Capability
  - Climb_Inhibit
- [SYSTEM-CHAP_03_EX]

	Cur_Vertical...	High_Confidence	Two_of_Thre...	Own_Tra...	..	Own_Tracke...	Alt_La
1	300	true	false	1		600	0
2	601	false	true	2		601	0
3	299	false	false	2		601	0
4	300	false	true	2		600	0
5	601	true	true	1		600	0
6	601	false	false	1		601	0
7	299	true	true	2		600	0
8	300	false	true	2		600	0
9	300	true	true	1		600	0
10	601	false	false	1		600	0
11	299	false	true	2		601	0
12	601	true	false	1		600	0
13	300	false	true	2		601	0
14	299	false	true	2		600	0
15	300	false	true	1		601	0
16	299	false	true	1		600	0
17	299	false	true	2		600	0
18	299	true	true	1		600	0
19	601	false	false	2		601	0
20	299	false	false	1		601	0
21	300	false	false	2		601	0

Figure 7-3. ACTS Main Window showing the SYSTEM-TACS

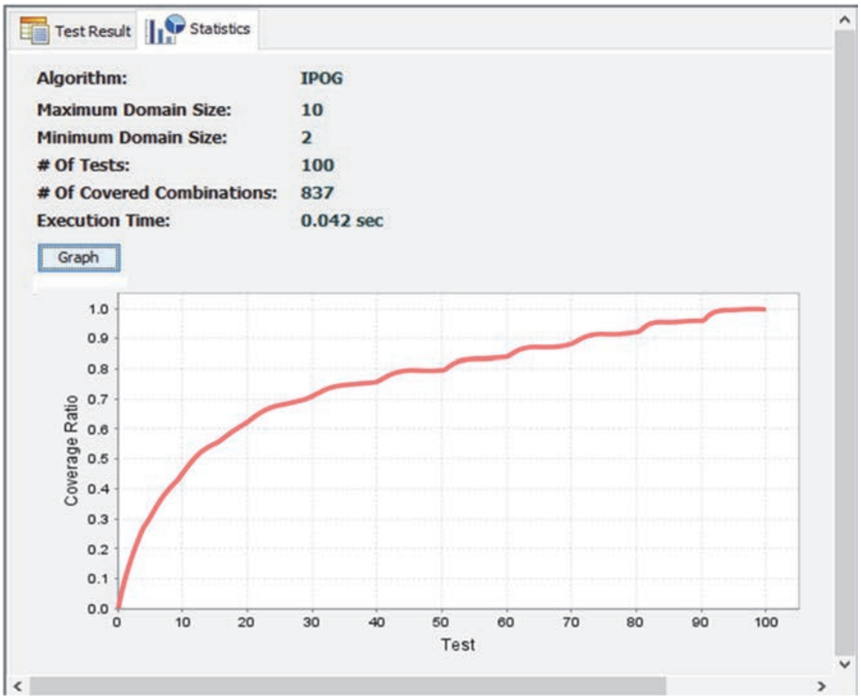


Algorithm: IPOG Strength: 2

[Root Node]
- [SYSTEM-TCAS]
  - Cur_Vertical_Sep
    - 299
    - 300
    - 601
  - High_Confidence
    - true
    - false
  - Two_of_Three_R
    - true
    - false
  - Own_Tracked_Alt

Algorithm:	IPOG
Maximum Domain Size:	10
Minimum Domain Size:	2
# Of Tests:	100
# Of Covered Combinations:	837
Execution Time:	0.042 sec

Figure 7-4. ACTS Main Window showing the Statistics tab.



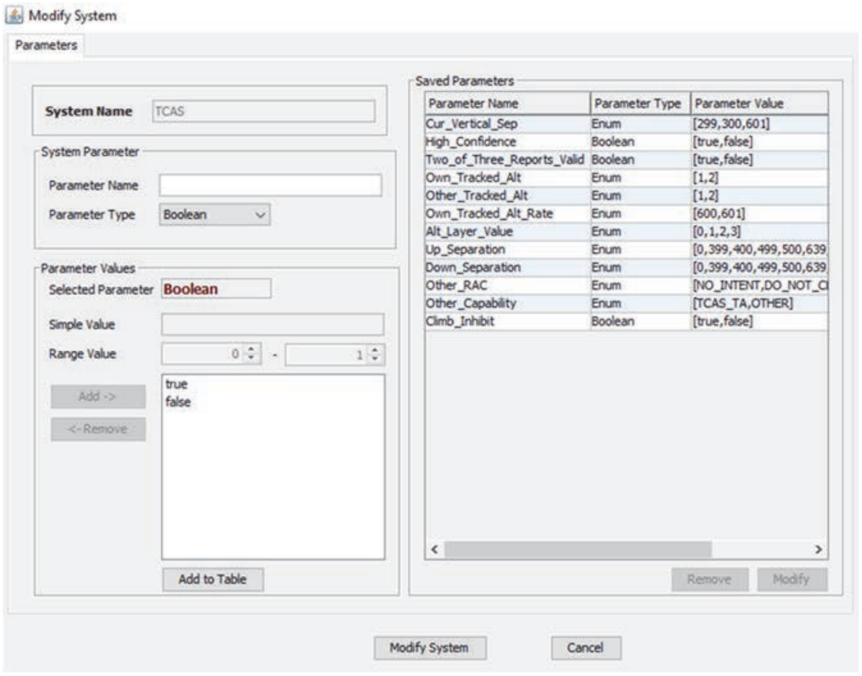
Algorithm: IPOG
Maximum Domain Size: 10
Minimum Domain Size: 2
Of Tests: 100
Of Covered Combinations: 837
Execution Time: 0.042 sec

Figure 7-5. ACTS Statistics window with graph

7.8 Modify System

To modify an existing system, select the system in the tree view, and then select menu ***Edit -> Modify***. The ***Modify System*** window (***Figure 7-6***) is the same as the ***New System*** window except that the name of the system cannot be changed. Note that the Modify System window is simplified in the Basic Version, e.g., it does not have the Constraints and Relations tabs.

A parameter can be added in the same way as in the New System window. A parameter can be removed by selecting the parameter from the Saved Parameters table on the right-hand side, and then clicking on the Remove button under the table. Note that a parameter cannot be removed if it is involved in a relation or constraint. In this case, the parameter must be removed from the involved relation or constraint first.



Parameter Name	Parameter Type	Parameter Value
Cur_Vertical_Sep	Enum	[299,300,601]
High_Confidence	Boolean	[true,false]
Two_of_Three_Reports_Valid	Boolean	[true,false]
Own_Tracked_Alt	Enum	[1,2]
Other_Tracked_Alt	Enum	[1,2]
Own_Tracked_Alt_Rate	Enum	[600,601]
Alt_Layer_Value	Enum	[0,1,2,3]
Up_Separation	Enum	[0,399,400,499,500,639
Down_Separation	Enum	[0,399,400,499,500,639
Other_RAC	Enum	[NO_INTENT,DO_NOT_C
Other_Capability	Enum	[TCAS_TA,OTHER]
Climb_Inhibit	Boolean	[true,false]

Figure 7-6. ACTS Modify window

7.9 Save/Save As/Save SUT As/Open System

To save an existing system, select the system in the tree view, and then select menu ***System -> Save*** or ***Save As***. When a newly created system is saved for the first time, or when ***Save As*** is selected, a standard file dialog will be brought up, where the user can specify the name of the file to be saved. The system will display a confirmation window if the file to be saved already exists. The ***Save SUT*** *As* menu option is used to save only the system configuration without the test set.

The user may open a system from either an XML or a TXT file and may save a system in either format too. To save a system in XML or TXT format, choose the *XML Files* or *Text Files* as file type in the Save System dialog as shown in ***Figure 7-7***.

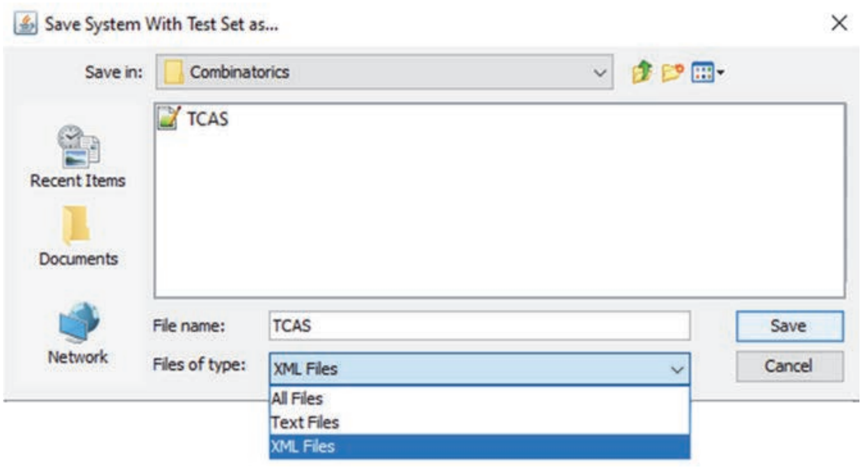


Figure 7-7. Save dialog with XML file-type selected.

7.10 Import/Export Test Set

To import a test set of a system, the user must first create the system configuration, in terms of adding into ACTS its parameters, values, relations and constraints, as described in **Section 7.1**. Then, select menu **Operations -> Import**, and select the format of the file containing the test set. Currently, two file formats are supported: CSV (default format), which stands for Comma Separated Values with the first row consisting of parameter names as column header, and CSV-RC, which stands for Comma Separated Values with Row and Column headers. (CSV-RC is mainly used to facilitate integration with Excel.) ***Figure 7-8*** shoes two example files, one for each format:

CSV format:

```
P1,P2,P3,P4,P5
0,2,2,3,6
3,2,4,2,2
2,1,2,1,3

3,2,5,0,5
```

CSV-RC format:

```
P1,P2,P3,P4,P5
Test1,0,2,2,3,6
Test2,3,2,4,2,2
Test3,2,1,2,1,3

Test4,3,2,5,0,5
```

Figure 7-8. Two file-types: CSV (left) and CSV-RC (right)

The parameter values in each row must be separated by ",". There can be arbitrary space between two values. After the file format is selected, a standard file selection window appears through which the user can browse through the system and select the file containing the test set to be imported.

To export a test set that exists in the GUI, first select the corresponding system so that the test set is displayed in the Test Result tab of the Main window, and then select **Operations -> Export**. Currently, three formats are supported, namely, NIST Format, Excel Format and CSV Format. A snippet of an exported test set in the NIST format is shown below:

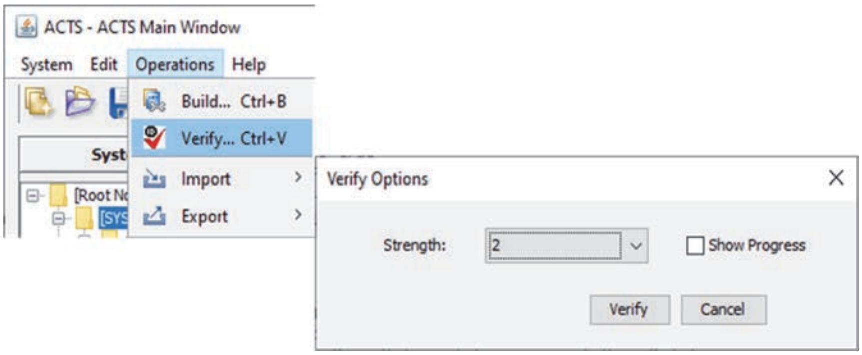
```
# ACTS Test Suite Generation: Tue May 19 06:45:46 CDT
2015
#  "(don't care)" represents don't care value
# Degree of interaction coverage: 2
# Number of parameters: 4
# Maximum number of values per parameter: 2
# Number of configurations: 6
-------------------Test Cases----------------
Configuration #1:
1 = P1=true
2 = P2=true
3 = P3=true
4 = P4=true
-----------------------------
Configuration #2:
```

7.11 Verify T-Way Coverage

This operation is typically used to verify the coverage of a test set that is imported from outside of ACTS. It will first find all possible combinations that need to cover, and then find how many combinations are actually covered by the current test set.

To verify the *t*-way coverage of a test set, the user can select the system in the System View, and then select menu ***Operations -> Verify***.

This brings up the *Verify Options* window, as shown in ***Figure 7-9***. If the "*Ignore Constraints*" option is checked, all the constraints specified in the system configuration will not be considered during verification. If the test strength is set to "*Mixed*", coverage will be verified against the relations specified in the system configuration. Note that this window is simplified in the Basic Version, where constraints and relations are not supported.



Strength: 2

Show Progress

Figure 7-9. Verify operations and options window

8 INCOMPLETE AND CONFOUNDED BLOCK DESIGNS

8.1 7.1 Introduction

One of the two purposes for randomized block designs, described in Chapter 4, was to group heterogeneous experimental units together in homogeneous subgroups called blocks. This increases the power or precision for detecting differences in treatment groups. The overall F-test for comparing treatment means, in a randomized block design, is a ratio of the variability among treatment means to the variability of experimental units within the homogeneous blocks.

One restriction on randomized block designs is that the number of experimental units in the block must be greater than or equal to the number of levels of the treatment factor. For the randomized complete block (RCB) design the number of experimental units per block, t, is equal to the number of levels of the treatment factor, and for the generalized complete block (GCB) design the number of experimental units per block is equal to the number of levels of the treatment factor times the number of replicates per block, tr.

When the number of levels of the treatment factor is large in a randomized block design, the corresponding number of experimental units per block must also be large. This could cause a problem. For example, frequently in agricultural field tests of new hybrid crops (called varietal trials) the number of hybrids could be very large (50-100). Since the experimental units are plots of land, the larger the number of plots included in a block the more diverse these plots will likely be since they cover a wider physical area. When experimental units are small animals, it is possible to group them into small blocks of littermates that are genetically more homogeneous. However, when larger block sizes are required, animals from different litters will be included within a block, making the groups less homogeneous. Less homogeneity within the blocks results in more variability among the experimental units within a block and defeats one of the purposes for using a randomized block design.

One solution to the problem is to construct block designs where each block only contains a subset of the possible levels of the treatment factor. In that way the number of experimental units per block, or block size, can be kept small. Designs that do this are called incomplete block designs. There are two common types of incomplete block designs. The first type is called a balanced incomplete block design or BIBD, and the other type is called a partially balanced incomplete block design or PBIB. Incomplete block designs are also useful when the physical requirements of experimentation make it impossible to test all levels of the treatment factor within each block of experimental units. Incomplete block designs are commonly used in agricultural experimentation, animal science, educational testing, and food science.

As an example of a situation where an incomplete block design would be useful, consider the following situation. In food science, taste panels are often used to test palatability of new recipes for new food products. In these panels, subjects are asked to taste and rate different recipes. The different recipes represent the levels of the treatment factor. A random sample of subjects (or the experimental units) is included to represent the potential market. Since there is wide variability in taste among subjects it is better to use a blocked design where each subject tastes and rates each recipe. However, if there are many recipes to be tested, subjects will lose their ability to discriminate, and it will be difficult to detect any differences. A solution is to have each subject (block) taste only a subset of the recipes.

8.2 Balanced Incomplete Block (BIB) Designs

Care must be taken when choosing the subset of treatment combinations tested in each block of an incomplete block design. If one treatment factor level is left out of every block, it cannot be compared to the other treatment levels. If different treatment levels are unequally represented, some comparisons of factor levels will have more precision than others.

The optimal way to create an incomplete block design is to have each treatment level equally replicated and appearing within a block with every other treatment level an equal number of times. This is called a balanced incomplete block design or BIB. By doing this, all pairwise differences of least squares treatment means will have the same standard error, and the power for detecting a difference in any two means will be the same.

The simplest way to construct a BIB for the case where there are t levels of the treatment factor and $k < t$ experimental units per block is to form all possible subsets of k treatment levels chosen from t. For example, in the taste panel described above, if there were $t = 6$ recipes to be tested, and it was determined that each subject could taste at most $k = 3$ recipes without losing discriminatory power, $\binom{6}{3} = 20$ subjects would be required. All possible subsets are listed below.

(1 2 3), (1 2 4), (1 2 5), (1 2 6), (1 3 4)
(1 3 5), (1 3 6), (1 4 5), (1 4 6), (1 5 6)
(2 3 4), (2 3 5), (2 3 6), (2 4 5), (2 4 6)
(2 5 6), (3 4 5), (3 4 6), (3 5 6), (4 5 6)

Thus, subject one would taste recipes 1, 2, and 3 in a random order; subject 2 would taste recipes 1, 2, and 4, and so forth. This plan is completely balanced in that each treatment level or recipe is replicated $r = 10$ times (or tasted by 10 subjects) and each pair of treatment levels occurs together in the same block $\lambda = 4$ times. For example, treatment levels 1 and 2 occur together only in the first four blocks on the first line above. By inspection, it can be seen that all pairs of treatment levels occur together in only four blocks.

Although taking all possible subsets of size k chosen from t is the simplest way to form a balanced incomplete block design, there may be other balanced incomplete block designs that require fewer blocks. If the precision of the design does not require as many as $\binom{b}{k}$ blocks, there would be no need to use that many. For example, if a practical size difference in recipes could be detected with less than 20 subjects and 10 replicates of each treatment level, in the taste test panel, perhaps a BIB design could be found with less than 20 blocks. If r is the number of times a treatment level is replicated in an incomplete block design, λ is the number of times each treatment level occurs with every other treatment level in the same block, t is the number of levels of the treatment factor, k is the number of experimental units in a block, and b is the number of blocks, then the following requirements must be met in order for that design to be a BIB design.

$$b \geq t \tag{8.1}$$
$$tr = bk \tag{8.2}$$
$$\lambda(t-1)\, r(k-1) \tag{8.3}$$

These relations can be used to find the minimum number of blocks required for a BIB design. Since r and λ must be integers, by Equation (8.3) we see that $\lambda(t-1)$ must be divisible by $k-1$. If $t=6$ and $k=3$, as in the taste test panel, 5λ must be divisible by 2. The smallest integer λ for which this is satisfied is $\lambda=2$. Therefore, it may be possible to find a BIB with $\lambda=2$, $r=10/2=5$, and $b=(6\times5)/3=10$. The function `BIBsize` in the **R** package `daewr` provides a quick way of finding values of λ and r to satisfy ***Equations (8.1)*** to ***(8.3)*** when given the number of levels of the treatment factor t and the block size k. For example, in the code below the function is called with $t=6$ and $k=3$.

```
> library(daewr)
> BIBsize(6, 3)
```

```
Possible BIB design with b = 10 and r=5 lambda=2
```

Fisher (1940) showed that even though Equations (8.1) and (8.3) are satisfied for some combination of t, b, r, λ, and k, a corresponding BIB may not exist. If a BIB does exist, Kiefer (1958) and Kshirsager (1938) have shown that it is D-optimal, thus it can be found using the R package `AlgDesign` that was described in Section 6.5.2. The function `optBlock` in that package can find BIB designs. For example, the code below searches for a BIB with $b=10$ blocks of $k=3$ experimental units per block and $t=6$ levels of the treatment factor. The option `"blocksizes"=rep(3,10)` specifies 10 blocks of size 3, and the option `"withinData"=factor(1:6)` specifies six levels of one factor.

```
> library(AlgDesign)
> BIB <- optBlock( ~ ., withinData = factor(1:6),
       blocksizes = rep(3, 10))
```

The object BIB created by optBlock contains a data frame BIB$design with one column that contains a list of treatment factor levels for each of the 10 blocks in order. Creating the design again with the optBlock function may result in a different but equivalent BIB.

```
> des <- BIB$rows
> dim(des) <- NULL
> des <- matrix(des, nrow = 10, ncol = 3, byrow = TRUE,
       dimnames = list(c( "Block1", "Block2", "Block3",
       "Block4", "Block5", "Block6", "Block7", "Block8",
       "Block9", "Block10"), c("unit1", "unit2",
       "unit3")))
> des
```

```
       unit1 unit2 unit3
Block1     4     5     6
Block2     2     3     5
Block3     2     3     6
Block4     1     2     5
Block5     1     2     4
Block6     1     5     6
Block7     2     4     6
Block8     1     3     4
Block9     3     4     5
Block10    1     3     6
```

According to this plan the three experimental units in the first block would receive treatment levels 4, 5, and 6, the experimental units in the second block would receive treatment levels 2, 3, and 5, and so forth. By inspection, it can be seen that each level of the treatment factor is repeated $r = 5$ times in this design, and that every treatment level occurs within the same block with every other treatment level $\lambda = 2$ times. Thus, we are assured that this is a balanced incomplete block design.

Once a BIB design is found, the levels of the treatment factor within each block should be randomized to the experimental units within that block, as illustrated for the RCB design in Section 4.2.

8.2.1 Two-Factor Factorial Experiments

This experimental design is not useful for identifying software problems by provides a simple illustration of the more difficult process of developing combinatorial designs.

To illustrate the analysis of a two-factor factorial experiment using the **R** function aov consider the data in ***Table 8-1***. These are the results of a two-factor experiment given by Hunter (1983). In this data, an experiment consisted of burning an amount of fuel and determining the CO emissions released. The experimental unit is the portion of a standard fuel required for one run, and the response, y, is the **carbon monoxide** (CO) emissions concentration in grams/meter^3 determined from that run. There were two replicate runs for each combination of factor levels separated by commas in ***Table 8-1***. Factor A is the amount of ethanol added to an experimental unit or portion of the standard fuel, and factor B is the fuel-to-air ratio used during the burn of that fuel.

Table 8-1. Data from Ethanol Fuel Experiment

A=ethanol additions	B=air/fuel ratio	y=CO emissions
0.1	14	66, 62
0.1	15	72, 67
0.1	16	68, 66
0.2	14	78, 81
0.2	15	80, 81
0.2	16	66, 69
0.3	14	90, 94
0.3	15	75, 78
0.3	16	60, 58

The data for this experiment is stored in the data frame `COdata` in the `daewr` package where the levels of ethanol and ratio are stored as the factors `Eth` and `Ratio`. The **R** commands to analyze the data are shown below.

```
> library(daewr)
> mod1 <- aov( CO ~ Eth * Ratio, data = COdata )
> summary(mod1)
```

The ANOVA table that results follows. There it can be seen that aov produces a table of the sums of squares, as described earlier. It can be seen from the tables that the two effects and their interaction are significant as indicated by the p-values to the right of the F-values.

```
            Df Sum Sq Mean Sq F value  Pr(>F)
Eth          2    324     162    31.4 8.8e-05 ***
Ratio        2    652     326    63.1 5.1e-06 ***
Eth:Ratio    4    678     170    32.8 2.2e-05 ***
Residuals    9     46       5
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 '
' 1
```

The model.tables function produces the results shown on the next page. The top line is the estimate of the overall mean $\hat{\mu}$. The next two sections show the marginal means for each factor along with the standard deviation of the values averaged in each mean. If the interaction was not significant, the marginal means would reveal the direction of the factor effects, but further preplanned comparisons or other multiple comparison procedures could be used to draw definite conclusions. The next section shows the cell means, and the final section shows the standard errors of the differences in marginal and cell means.

```
> model.tables( mod1, type = "means", se = T )

Tables of means
Grand mean

72.833

 Eth
Eth
  0.1   0.2   0.3
66.83 75.83 75.83

 Ratio
```

```
Ratio
  14   15   16 
78.5 75.5 64.5 

 Eth:Ratio
     Ratio
Eth   14   15   16  
  0.1 64.0 69.5 67.0
  0.2 79.5 80.5 67.5
  0.3 92.0 76.5 59.0

Standard errors for differences of means
          Eth Ratio Eth:Ratio
        1.312 1.312     2.273
replic.     6     6         2
```

Two estimate specific contrasts of the main effects, the estimable function from the **R** package gmodels can be utilized. To use it we must first construct contrasts to replace the default treatment contrasts used by the **R** function aov. For example, in the first statement below we construct the contrast coefficients for comparing the first factor level to the third in a three-level factor. A second contrast orthogonal to the first is also constructed, and the contrast matrix cm is created by using the two contrasts as columns.

```
> c1 <- c(-1/2, 0, 1/2)
> c2 <- c(.5,-1, .5)
> cm <- cbind( c1, c2 )
```

In the call of the aov function below, the cm contrast matrix will be used for both main effects rather than the default treatment contrasts used by the aov function. The next lines load the gmodels package and create labels for the contrasts which compare the first factor level to the third factor level. The vector following each label is an indicator vector for which model coefficient is displayed. It selects the first coefficient for ethanol and ratio. Finally, the estimable function is called with the inputs being the mod2, that was created by the aov function, and the contrast labels and definitions in c.

```
> mod2 <- aov( CO ~ Eth * Ratio, contrasts = list( Eth =
      cm, Ratio = cm ), data = COdata)
> library(gmodels)
> c <- rbind( 'Ethanol 0.3 vs 0.1' =
      c(0,1,0,0,0,0,0,0,0),'Ratio  16  vs  14'  =
      c(0,0,0,1,0,0,0,0,0)  )
> estimable(mod2,c)
```

```
                 Estimate Std. Error t value DF   Pr(>|t|)
Ethanol 0.3 vs 0.1      9     1.3123   6.858  9 7.4066e-05
Ratio  16  vs  14     -14     1.3123 -10.668  9 2.0837e-0
```

These estimates would be estimable and meaningful if there were no significant interaction between ethanol addition level and air/fuel ratio, but in this case there is a significant interaction and the difference in CO emissions caused by changing the amount of ethanol addition will depend on the air/fuel ratio and the difference in CO emission caused by changing the air/fuel ratio will depend on the amount of ethanol added. An interaction graph is a better way of interpreting these results. An interaction plot can be generated using the R function ggplot as shown below. This code uses the aesthetics (aes) function using variables names in the data frame COdata to produce ***Figure 8-1***.

```
> library(ggplot2)
> ggplot(COdata, aes(x = Ethanol, y = CO,
                   shape = Ratio,
                   group = Ratio,
                   color = Ratio)) +
  stat_summary(fun = "mean", geom = "point", size = 3) +
  stat_summary(fun = "mean", geom = "line", size = 1.2) +
  labs(    title = "Interaction Plot of Ethanol and
      air/fuel ratio") +
  scale_color_manual(values = c("dodgerblue", "red",
      "darkgreen")) +
  theme_classic()
```

In this plot we can see more clearly the dependence of effects. Increasing the amount of ethanol added to the fuel from 0.1 to 0.3 causes CO emissions to increase linearly from 64 grams/liter to 92 grams/liter when the air/fuel ratio is at its low level of 14. This is shown by the blue line with circles representing the cell means. However, when the air/fuel ratio is at its high level of 16 (illustrated by the green line with squares representing the cell means), increasing the ethanol added to the fuel from 0.1 to 0.3 actually causes a decrease in CO emissions from 67 grams/liter to 59 grams/liter along a nearly linear trend. Finally, when the air/fuel ratio is held constant at its mid-level of 15 (illustrated by the red line with triangles representing the cell means), increasing ethanol from 0.1 to 0.2 causes CO emissions to increase by 11 grams/liter; but a further increase in ethanol to 0.3 causes a decrease in CO emissions of 4 grams/liter to 76.5.

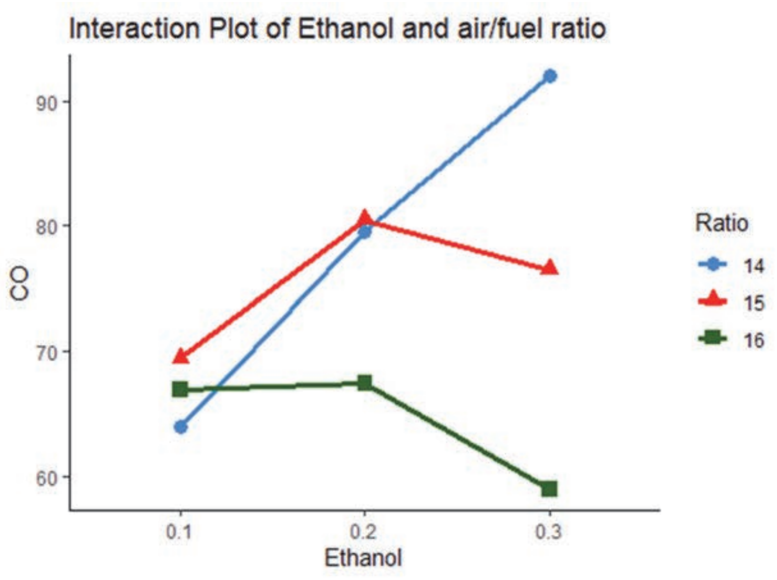


Figure 8-1. Interaction Plot Ethanol and Air/Fuel Ratio

The interpretation above again illustrates the principle of comparing the effect of one factor across the levels of the other factor in order to describe an interaction. This was done by comparing the effect of changing the ethanol addition between the levels of air/fuel ratio. It could also be done in the opposite way. For example, the R code below reverses the interaction plot as shown in ***Figure 8-2***.

```
> ggplot(COdata, aes(x = Ratio, y = CO,
                     shape = Ethanol,
                     group = Ethanol,
                   color = Ethanol)) +
  stat_summary(fun = "mean", geom = "point", size = 3) +
  stat_summary(fun = "mean", geom = "line", size = 1.2) +
  labs(    title = "Interaction Plot of Ethanol and
       air/fuel ratio") +
  scale_color_manual(values = c("blue", "darkorange",
       "green")) +
  theme_classic()
```

In this plot the green line, with squares representing the cell means, shows the effect of increasing air/fuel ratio when ethanol is added at the high rate of 0.3. Carbon monoxide emissions decrease linearly from 92 grams/liter to 59 grams/liter. However, when ethanol is added at the low rate of 0.1, the CO emissions actually increase slightly from 64 grams/liter to 67 grams/liter as a result of increasing air/fuel ratio from 14 to 16. This can be seen on the blue line with circles representing the cell means. When ethanol is added at the mid-rate of 0.2, there is little change in CO emissions when air/fuel ratio is increased from 14 to 15, but there is a decrease in CO emissions of 13 grams/liter caused by increasing air/fuel ratio from 15 to 16. The latter result can be visualized on the orange line with triangles representing the cell means.

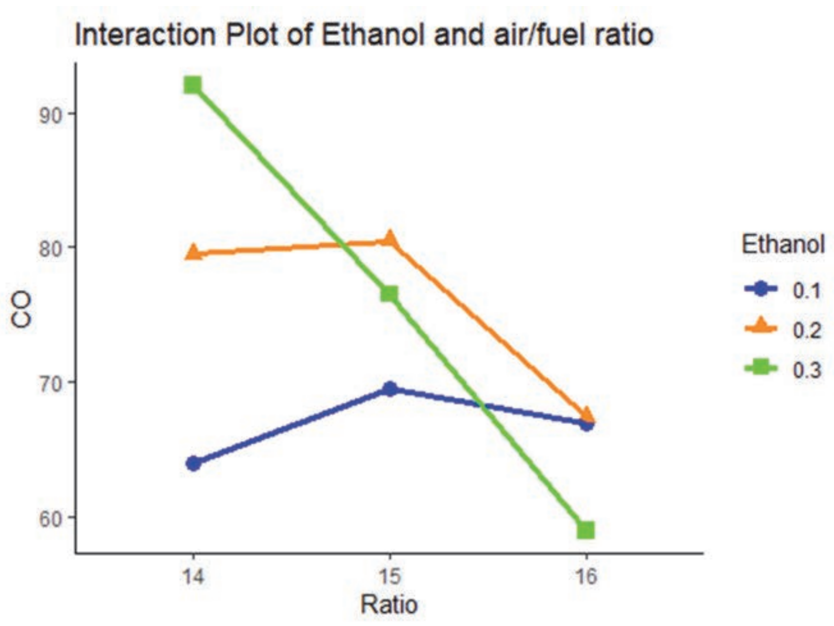


Figure 8-2. Interaction Plot Ethanol and Air/Fuel Ratio

Either way of presenting and interpreting the interaction is valid as long as one discusses how the effect of one factor changes depending upon the level of the other. The factor effects, that should be compared depend on which one is of most interest in a particular research problem. Another thing to notice about the two interpretations is that cause and effect relationships are assumed. We say the change in the response is caused by the change in the factor or the change in the response is the result of changing the factor. This statement could not be made when discussing the results of an observational study.

8.2.2 Determining the Number of Replicates

One of two possible methods can be followed to determine the number of replicates for a factorial experiment that will result in a power between 0.80 to 0.90 (for detecting differences that have practical significance). The first method is to consider detecting differences among the cell means. The second method is to consider detecting a practical size difference in the marginal means for the factors in the experiment.

When looking for differences among the cell means, the cells in the factorial are considered to be an unstructured group as in a one-factor design. Using the cell means model $y_{ijk} = \mu_{ij} + \varepsilon_{ijk}$ the procedure is the same as it was described for the one-factor model $y_{ij} = \mu_i + \varepsilon_{ij}$. The noncentrality parameter for the *F*-test is:

$$\lambda = \left(\frac{r}{\sigma^2}\right) \sum_{i-1}^{a} \sum_{j=1}^{b} \left(\bar{\mu}_{ij} - \bar{\mu}_{..}\right)^2$$

When looking for differences in the marginal means for the factors, the non- centrality factor for the first main effect is:

$$\lambda_a = br \sum_i \frac{\alpha_i^2}{\sigma^2} = br \sum_i \frac{(\bar{\mu}_{i.} - \bar{\mu}_{..})^2}{\sigma^2}$$

and for the second main effect the noncentrality factor is:

$$\lambda_b = ar \sum_i \frac{\beta_i^2}{\sigma^2} = ar \sum_i \frac{\left(\bar{\mu}_{.j} - \bar{\mu}_{..}\right)^2}{\sigma^2}$$

If Δ is considered to be the size of a practical difference in cell means, then the smallest $\lambda = r/\sigma^2 \sum_{i=1}^{a} \sum_{j=1}^{b} (\bar{\mu}_{i.} - \bar{\mu}_{..})^2$ could be with two cells differing by at least Δ is $r\Delta^2/2\sigma^2$ Likewise, if Δ is considered to be the size of a practical difference in marginal means for factor A, the smallest $\lambda_a = br \sum_i (\bar{\mu}_{i.} - \bar{\mu}_{..})^2/\sigma^2$ could be with two marginal means differing by at least Δ is $br\Delta^2/2\sigma^2$. Here again we can see the efficiency of factorial designs because the noncentrality factor for detecting differences in marginal factor A means is larger than the noncentrality factor for detecting differences of cell means by a factor of *b*, the number of levels of factor B.

Consider the following example. A paper helicopter experiment is planned to investigate the effects of four levels of factor A = wing length, and four levels of factor B = body width, upon the flight time. If pilot experiments with nine replicates of one design resulted in flight times of 2.8, 2.6, 3.5, 3.0, 3.1, 3.5, 3.2, 3.4, and 3.4 seconds. How many replicates would be required to detect a difference in flight times of 1 second with a power of 0.90?

From the pilot tests the standard deviation of experimental error can be estimated as $s = 0.32$. If $\Delta = 1.0$ is considered to be a practical size difference in cell means, we use the **R** code below to give the answer. In a 4-by-4 factorial there are 16 cells so the number of levels of the factor is considered to be 16. The modified **R** code is:

```
> library(daewr)
> rmin <- 2 # smallest number of replicates
> rmax <- 8 # largest number of replicates
> alpha <- .05
> Delta <- 1
> nlev <- 16
> nreps <- c(rmin:rmax)
> power <- Fpower1(alpha, nlev, nreps, Delta, sigma)
> options(digits  =  5)
> power
```

The results of running this code show that 6 replicates per cell would be required to obtain a power of at least 0.90.

```
      alpha   nlev   nreps   Delta   sigma   power
[1,]  0.05    16     2       1       0.32    0.24173
[2,]  0.05    16     3       1       0.32    0.48174
[3,]  0.05    16     4       1       0.32    0.69246
[4,]  0.05    16     5       1       0.32    0.83829
[5,]  0.05    16     6       1       0.32    0.92326
[6,]  0.05    16     7       1       0.32    0.96664
[7,]  0.05    16     8       1       0.32    0.98655
```

If $\Delta = 1.0$ is considered to be a practical size difference in marginal means for one of the factors, the results will be different. The degrees of freedom for the numerator would be $\nu_1 = 4 - 1$, the degrees of freedom for the denominator would be $\nu_2 = 16(r - 1)$, the noncentrality factor for a main effect A would be $\lambda_a = br\Delta^2/2\sigma^2$, and the noncentrality factor for a main effect B would be $\lambda_b = ar\Delta^2/2\sigma^2$. The **R** code below demonstrates the use of the Fpower2 function in the daewr package for determining the number of replicates required to detect a difference of Δ in marginal means of the factors in a two-factor factorial. The arguments that must be supplied to Fpower2 are: alpha, nlev (a vector of length 2 containing the number of levels of the first factor (A) and the second factor (B)), nreps $= r$, Delta $= \Delta$, and sigma $= \sigma$.

```
> library(daewr)
> rmin <- 2 # smallest number of replicates
> rmax <- 4 # largest number of replicates
> alpha <- .05
> sigma <- .32
> Delta <- 1.0
> nlev <- c(4,4)
> nreps <- c(rmin:rmax)
> result <- Fpower2(alpha, nlev, nreps, Delta, sigma)
> options(digits = 5)
> result
```

The results of running this code appear below. Here it can be seen that with only $r =$ two replicates per cell the power for detecting a $\Delta = 1.0$ difference in marginal means for factor A or B is greater than the power for detecting differences of $\Delta = 1.0$ in cell means with $r = 8$ replicates per cell. Again this demonstrates the efficiency of factorial experiments through hidden replication.

```
     alpha a b nreps Delta sigma  powera  powerb
[1,]  0.05 4 4     2     1  0.32 0.99838 0.99838
[2,]  0.05 4 4     3     1  0.32 1.00000 1.00000
[3,]  0.05 4 4     4     1  0.32 1.00000 1.00000
```

With the ability to calculate power quickly, it is possible to explore many potential designs before actually running the experiments. The number of factors, the number of levels of each factor, and the number of replicates in each cell all affect the power to detect differences. Power calculations help an experimenter to determine an efficient use of his or her resources.

8.2.3 Analysis with an Unequal Number of Replicates per Cell

Although it would be unusual to plan a factorial experiment with an unequal number of replicates per cell, the data from a factorial experiment may end up with an unequal number of replicates due to experiments that could not be completed, or responses that could not be measured, or simply lost data. As long as the chance of losing an observation was not related to the treatment factor levels, the data from a factorial experiment with an unequal number of replicates per cell can still be analyzed and interpreted in a manner similar to the way it would be done for the equal replicate case. However, the computational formulas for analyzing the data differ for the case with an unequal number of replicates.

To illustrate why the analysis shown in Section 8.2.1 is inappropriate, con- sider again the data from the ethanol fuel experiment described in Section 8.2.1. This time assume one observation in the cell where air/fuel ratio = 16 and ethanol level = 0.3 was missing. Then ***Table 8-2*** shows the data with each response value written above its symbolic expected value. The **R** code below the table creates a data frame containing the data in ***Table 8-2***, by inserting a missing value into the 18th row and third column.

Table 8-2. Fuel Experiment with Unequal Reps

Ethanol	air/fuel 14	air/fuel 15	air/fuel 16
0.1	66 62 $\mu+\alpha_1+\beta_1+\alpha\beta_{11}$	72 67 $\mu+\alpha_1+\beta_2+\alpha\beta_{12}$	68 66 $\mu+\alpha_1+\beta_3+\alpha\beta_{13}$
0.2	78 81 $\mu+\alpha_2+\beta_1+\alpha\beta_{21}$	80 81 $\mu+\alpha_2+\beta_2+\alpha\beta_{22}$	66 69 $\mu+\alpha_2+\beta_3+\alpha\beta_{23}$
0.3	90 94 $\mu+\alpha_3+\beta_1+\alpha\beta_{31}$	75 78 $\mu+\alpha_3+\beta_2+\alpha\beta_{32}$	60 $\mu+\alpha_3+\beta_3+\alpha\beta_{33}$

```
> COdatam <- COdata
> COdatam[18, 3] <- NA
```

The marginal column means for the levels of air/fuel ratio factor computed using the `model.tables` statement as shown on page 66 and the modified data frame `COdatam` would be 78.5, 75.5, and 65.8, respectively. The expected value of the marginal means for the first two columns would be: $\mu + \beta_1$, $\mu + \beta_2$, since $(\alpha_1 + \alpha_2 + \alpha_3)/3 = 0$ and $(\alpha\beta_{1i} + \alpha\beta_{2i} + \alpha\beta_{3i})/3 = 0$ for $i = 1{,}2$. However, the expected value of the last marginal column mean would be $\mu + \beta_3 + (2\alpha_1 + 2\alpha_2 + \alpha_3)/5 + (2\alpha\beta_{13} + 2\alpha\beta_{23} + 2\alpha\beta_{33})/5$ and is not an unbiased estimate of $\mu + \beta_3$. The comparison between the first and third column means would not be an unbiased estimate of $\beta_1 - \beta_3$. Likewise, the last marginal row mean would not be an unbiased estimate of $\mu + \alpha_3$.

If the ANOVA table of the data in `COdatam` is produced with the R function `aov`, the *F*-tests will not test the same hypotheses that they do in the case of equal number of replicates in the cells. When there is an unequal number of replicates in the cells, the noncentrality parameter for the *F*-test of H_0: $\alpha_1 = \ldots = \alpha_a$, that is based on $R(\alpha|\mu)$ will not be $\lambda_a = rb\sum_i a_i^2$ but a quadratic form involving the elements of α, β as well as $\alpha\beta$. The noncentrality for the *F*-test test of H_0: $\beta_1 = \ldots = \beta_b$ based on $R(\beta|\mu, \alpha)$ will be a quadratic form involving the elements of β and $\alpha\beta$.

To calculate adjusted sums of squares for the null hypothesis for the main effects, use the `contr.sum` option in the **R** `lm` function and the Anova function from the **R** package car (Fox and Weisberg, 2011). The option type II in the Anova function computes the type II sums of squares, and the option type III produces the type III sums of squares. The type II sum of squares for the factors A and B can be represented as $ssA_{II} = R(\alpha|\mu, \beta)$, and $ssB_{II} = R(\beta|\mu, \alpha)$. $R(\alpha|\mu, \beta)$ is the difference in the error sums of squares for the reduced model where $X = (1|X_B)$ and the full model where $X = (1|X_A|X_B|X_{AB})$. The corresponding noncentrality factor for the corresponding *F*-test will be a quadratic form that only involves $\alpha' = (\alpha_1, \alpha_2, \alpha_3)$ and $\alpha\beta'$. When there is an equal number of replications per cell, the sums of squares computed by the aov function are identical to the type II sums of squares.

The type III sum of squares for the factors A and B can be represented as $ssA_{III} = R(\alpha|\mu, \beta, \alpha\beta)$, and $ssB_{III} = R(\beta|\mu, \alpha, \alpha\beta)$. $R(\alpha|\mu, \beta, \alpha\beta)$ is the difference in the error sums of squares for the reduced model where $X = (1|X_B|X_{AB})$ and the full model where $X = (1|X_A|X_B|X_{AB})$. The corresponding noncentrality factor for the corresponding *F*-test will be a quadratic form that only involves $\alpha' = (\alpha_1, \alpha_2, \alpha_3)$. When there is an equal number of replications per cell, the sums of squares computed by the aov function are identical to the type III sums of squares.

Some analysts prefer to use the type II sums of squares and others prefer the type III sums of squares when there is an unequal number of replicates per cell. In this book we illustrate the type III sums of squares and hypothesis tests, although the type II sums of squares can be obtained by changing the option from type = "III" to type = "II" in the call to the Anova function.

The code to produce the type III ANOVA table ethanol fuel experiment after removing the observation with the value of 58 (from the cell with the air/fuel ratio = 16 and the ethanol level = 0.3) is shown below.

```
> library(car)
> mod2 <- lm( CO ~ Eth*Ratio, data = COdatam, contrasts
+ = list( Eth = contr.sum, Ratio = contr.sum ))
> Anova( mod2, type="III" )
```

The results are below.

```
Anova Table (Type III tests)

Response: CO
            Sum Sq Df F value  Pr(>F)
(Intercept)  86198  1 15496.4 1.9e-14 ***
Eth            319  2    28.7 0.00022 ***
Ratio          511  2    46.0 4.1e-05 ***
Eth:Ratio      555  4    24.9 0.00014 ***
Residuals       44  8
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘
’ 1
```

In order to get means that have expectations $\mu + \beta_1$, $\mu + \beta_2$, $\mu + \beta_3$ when there are an unequal number of replicates per cell, the adjusted means should be computed. The adjusted means (sometimes called least-squares means or `lsmeans` for short) are computed by calculating the marginal means of the predicted cell means, $\left(\hat{\mu} + \hat{\alpha}_i + \hat{\beta}_j + \widehat{\alpha\beta}_{ij}\right)$ obtained from the least squares estimates of the model parameters. Remember that the cell means are estimable functions. The **R** code below calculates the predicted cell means using the effect estimates from the model `mod2` created by the `lm` function shown above and then computes the adjusted or `ls` marginal means for the air/fuel ratio using the **R** function `tapply`.

```
> p <- data.frame( expand.grid( Eth = c(.1, .2, .3), Ratio
= c(14,15,16) ) )
> p[] <- lapply(p, factor)
> p <- cbind( yhat = predict( mod2, p), p)
> with(p, tapply(yhat, Ratio, mean) )
```

```
    14      15      16
78.500 75.500 64.833
```

In these results it can be seen that the means for the first two columns are the same as the simple arithmetic average of the responses in the first two columns as shown on page 66, but the mean from the third column is different, and it is a more accurate estimate of $\mu + \beta_3$. The **R** package `lsmeans` automatically computes the adjusted or `lsmeans`, and in addition it computes their standard errors and confidence limits. The **R** code below illustrates the use of this package to compute the marginal adjusted means for both ethanol and air/fuel ratio. The NOTE: printed by the `lsmeans` function tells us what we already know: interpretation of the marginal means may be misleading when there is a significant interaction.

```
> library(lsmeans)
> lsmeans(mod2,~ Eth)
```

```
NOTE: Results may be misleading due to involvement in
interactions
 Eth lsmean    SE df lower.CL upper.CL
 0.1    66.8 0.963  8     64.6     69.1
 0.2    75.8 0.963  8     73.6     78.1
 0.3    76.2 1.112  8     73.6     78.7

Results are averaged over the levels of: Ratio
Confidence level used: 0.95

> lsmeans(mod2,~Ratio)

NOTE: Results may be misleading due to involvement in
interactions
 Ratio lsmean    SE df lower.CL upper.CL
 14      78.5 0.963  8     76.3     80.7
 15      75.5 0.963  8     73.3     77.7
 16      64.8 1.112  8     62.3     67.4

Results are averaged over the levels of: Eth
Confidence level used: 0.95
```

In general the type II or III sums of squares and `lsmeans` should be used, because they will test the correct hypotheses and provide unbiased factor level means whether there is an equal or unequal number of replications per cell.

8.3 Analysis of Incomplete Block Designs

The model for an incomplete block design is

$$y_{ij} = \mu + b_i + \tau_j + \varepsilon_{ij} \tag{7.4}$$

which is identical to the model for a randomized complete block design given in Section 8.4. However, the analysis is slightly different due to the missing observations.

8.3.1 An Example

Consider the data from a taste panel experiment reported by Moskowitz (1988), shown in ***Table 7.1***. This experiment is a BIB with $t = 4$ levels of the treatment factor or recipe, and block size $k = 2$.

Thus each panelist tastes only two of the four recipes in a random order and assigns a category scale score. Category scales are commonly used in assessing food likes or dislikes and consist of numbers 1 to 10 that represent descriptive categories. Only $\binom{4}{2} = 6$ blocks or panelists are required for a BIB, but in this experiment that number was doubled in order to increase the power for detecting differences. Thus the first six panelists and the last six are a repeat of the same BIB design. Subjects participating in the taste panel were randomly assigned to panelist numbers and the order of the two recipes tasted by each panelist was randomized.

Table 8-3. Data from BIB Taste Test

	Recipe			
Panelist	A	B	C	D
1	5	5	-	-
2	7	-	6	-
3	5	-	-	4
4	-	6	7	-
5	-	6	-	4
6	-	-	8	6
7	6	7	-	-
8	5	-	8	-
9	4	-	-	5
10	-	7	7	-
11	-	6	-	5
12	-	-	7	4

When analyzing the data from an incomplete block design, the marginal treatment means are not unbiased estimators of the estimable effects $\mu + \tau_i$. For example, in ***Table 8-3*** the marginal mean for recipe A could be biased low by the fact that it was not tasted by panelists 4, 5, 6, 10, 11, and 12 who seem to rate recipes higher. Likewise, the noncentrality factor for the sequential sums of squares for treatments (or recipes) may contain block effects as well as treatment effects. The solution to these problems is the same as shown in ***Section 8.2.2*** for analyzing data from factorial designs with an unequal number of replicates per cell. The least squares means are used rather than marginal means, and the type III or adjusted sums of squares for treatments should be used. The Anova function in the car package can be used to get the type III or adjusted sums of squares for recipes as shown in ***Section 8.2.2***, or the `lm` function can be used with the recipe term ordered last in the model as shown in the code below. `lm` will not produce the type III or adjusted sums of squares for blocks or panelists, but since differences in panelists are not of interest they are not needed.

```
> library(daewr)
> mod1 <- aov( score ~ panelist + recipe, data = taste)
> summary(mod1)
```

```
Df Sum Sq Mean Sq F value Pr(>F)
panelist 11 19.333 1.7576 2.301 0.1106
recipe 3 9.125 3.0417 3.982 0.0465 *
Residuals 9 6.875 0.7639
---
Signif. codes: 0 *** 0.001 ** 0.01 * 0.05 . 0.1 1
```

The $F_{3,9}$ value for testing recipes was 3.982, which is significant at the $\alpha = 0.05$ significance level. The adjusted or least squares means can be calculated using the predict command and the `tapply` function as shown in ***Section 8.2.3***, but it can be done with less coding using the lsmeans package. The code below using lsmeans computes the adjusted means, their standard errors and confidence intervals, and makes Tukey comparisons of all pairs of means.

```
> library(lsmeans)
> lsmeans(mod1, pairwise ~ recipe, adjust = ("tukey"))
```

```
$lsmeans
 recipe lsmean    SE df lower.CL upper.CL
 A        5.46 0.418  9     4.51     6.40
 B        6.21 0.418  9     5.26     7.15
 C        6.83 0.418  9     5.89     7.78
 D        4.83 0.418  9     3.89     5.78

Results are averaged over the levels of: panelist
Confidence level used: 0.95

$contrasts
 contrast estimate    SE df t.ratio p.value
 A - B      -0.750 0.618  9  -1.214  0.6342
 A - C       -1.375 0.618  9  -2.225  0.1882
 A - D        0.625 0.618  9   1.011  0.7472
 B - C       -0.625 0.618  9  -1.011  0.7472
 B - D        1.375 0.618  9   2.225  0.1882
 C - D        2.000 0.618  9   3.236  0.0421

Results are averaged over the levels of: panelist
P value adjustment: tukey method for comparing a family of
4 estimates
p values are adjusted using the tukey method for 4 means
```

The results show the average score for recipe C is significantly higher than the score for recipe D at the $\alpha = 0.05$ level, but that no other differences are significant at the $\alpha = 0.05$ level. Another thing that should be noticed in the output is the fact that the standard error of the differences in means is the same for all pairs of means. This is a result of the fact that a BIB design was used.

8.4 Determining the Number of Blocks

The *F*-test for treatment or dose effect in the last example was highly significant $(P < 0.0001)$. If the experiment were to be repeated in a situation where the variability of the response (lever press rate within a rat) and differences in treatment means were to remain approximately the same, fewer blocks or rats would be required to detect significant differences in treatments. The no noncentrality parameter for the *F*-test of treatment effects in the randomized complete block design is $\lambda = \left(\frac{b}{\sigma^2}\right)\sum_j \tau_j^2$, and the degrees of freedom are $\nu_1 = t - 1$, and $\nu_2 = (b-1)(t-1)$. Therefore, in order to calculate the number of blocks that will result in a power between 0.8 and 0.9 for detecting a difference in treatment means, the **R** code in Section 8.2.2 can be modified by changing the formula for the denominator degrees of freedom and the noncentrality factor.

The **R** code below can be used for calculating the power for a randomized block design as a function of the number of blocks. Using the results from the last experiment, the estimate of $\sigma^2_{rcb} = 0{:}00834867$ and $css = \sum_j \tau_j^2$ can be estimated to be

$$(0.764 - 0.9142)^2 + \ \dots \ + \ (0.850 - 0.9142)^2 = 0.460208$$

Using these as inputs to a more general F-power function that takes the degrees of freedom, and the noncentrality parameter as arguments, the results are created that are shown below the code. There it can be seen that a power greater than 0.99 can be achieved with only $b = 2$ blocks or rats in the experiment.

```
> library(daewr)
> bmin <- 2
> bmax <- 3
> alpha <- .05
> sigma2 <- 0.0083487
> css <- 0.460208
> nu1 <- 5-1
> blocks <- c(bmin:bmax)
> nu2 <- (blocks - 1) * 4
> nc <- (blocks * css) / sigma2
> Power <- Fpower( alpha, nu1, nu2, nc )
> data.frame(blocks, nu1, nu2, nc, Power)
```

```
  Blocks   nu1   nu2          nc      Power
1    2       4     4    110.2466  0.9966799
2    3       4     8    165.3699  1.0000000
```

If an estimate σ^2_{crd} were available from previous experiments or pilot studies, Hinkelmann and Kempthorne (1994) have shown the relative efficiency (RE) σ^2_{crd} can also be used to get a rough estimate of the number of blocks required for an RCB design. For example, suppose σ^2_{crd} were estimated to be 0.040865 from previous experiments and the number of replicates of each treatment required for a CRD design to achieve adequate power for detecting a practical difference in means was $r = 20$, determined by the methods of Section 8.2.2. If blocking was expected to reduce the variance by 90% (i.e., $\sigma^2_{rcb} = 0{:}\,10 \times \sigma^2_{crd}$, or $RE = 10.0$). Then the number of blocks required to achieve the same power with an RCB design is $b = \frac{r}{RE} = \frac{20}{10} = \ 2$.

8.4.1 Determining the Number of Replicates

A rough estimate of the number of replicates of each treatment level, $r = \frac{bk}{t}$, or the number of blocks, $b = \frac{tr}{k}$, required for adequate power for detecting a practical difference in treatment means in a BIB design can be determined using the following strategy. If an estimate of σ^2, the variance of heterogeneous experimental units, and Δ, the size of a practical difference in two treatment means are available, then use the method of Section 8.2.2 to determine the number of replicates, r_{crd}, required for a completely randomized design. If blocking the heterogeneous experimental units into small blocks of size k is expected to reduce the variance of experimental units within the blocks by a percentage equal to $100 \times \left(1 - \frac{1}{RE}\right)$, then following Section 8.4 the number of replicates required for the blocked design would be $r = \frac{r_{crd}}{RE}$ and the number of blocks of size k in an incomplete block would be $b = \frac{tr}{k}$.

8.5 BTIB and PBIB Designs

Sometimes a BIB design requires more blocks or experimental units than are available or needed to obtain adequate power for detecting practical differences in treatment means. In this situation, the number of blocks and experimental units can be reduced by relaxing the requirements that: (1) each treatment level be equally replicated and (2) that it appears within a block with every other treatment level the same number of times. By relaxing these requirements, each treatment level will occur more frequently in a block with some treatment levels than it will with other treatment levels. Therefore some pair-wise comparisons of treatment means will have smaller standard errors than others.

In other situations, a BIB design cannot be used because physical constraints of experimentation prevent some treatment levels from being tested together within the same block. For example, an experiment was conducted to compare the reading of a portable home use blood pressure monitor to other automatic blood pressure monitors in supermarket pharmacies in order to determine if readings from the portable monitor were biased high. The experimental unit was the pressure in a subject's veins at the time it was measured, and the treatment factor levels were the monitors used to make the measurement.

Blood pressure is known to vary widely from person to person and within a person at different times throughout the day. Blood pressure is most consistent in one person within a short period of time. Therefore the experimental units were blocked into homogeneous groups by subject and time. The portable blood pressure monitor could be carried into a store and a subject's blood pressure could be checked within a short period of time by both the automatic monitor within the store and the portable monitor. However, the monitors from two stores could not be compared within a block, or for a short period of time, because the stores were physically separate. Driving between stores could completely change a subject's blood pressure, so the incomplete block design shown in ***Table 8-4*** was utilized. The response (diastolic blood pressure) is shown in the table.

Table 8-4. Incomplete Block Design with Blood Pressure Monitors

	Treatment			
Block	Portable Monitor	Store A Monitor	Store B Monitor	Store C Monitor
1=(subject 1, time 1)	85	77	-	-
2=(subject 2, time 1)	80	75	-	-
3=(subject 1, time 2)	89	-	73	-
4=(subject 2, time 2)	80	-	70	-
5=(subject 1, time 3)	78	-	-	76
6=(subject 2, time 3)	80	-	-	70

Here it can be seen that treatment level 1 (portable monitor) appears in a block with every other treatment level, but the other treatment levels never appear together in a block. The code to analyze the data using **R** `aov` function and the `lsmeans` function is shown below.

```
> library(daewr)
> modm <- aov( pressure ~ Block + Treatment, data =
       BPmonitor)
> library(lsmeans)
> lsmeans(modm,pairwise~Treatment,adjust=("tukey"))
```

In the resulting table of comparisons of means, shown at the top of the next page, it can be seen that the standard errors of the differences between the portable monitor (P) mean and the means for the other monitors (A, B, and C) are smaller than the standard errors of the comparisons between A and B, A and C, and B and C.

```
$lsmeans
 Treatment lsmean   SE df lower.CL upper.CL
 A            75.5 2.75  3     66.7     84.3
 B            69.0 2.75  3     60.2     77.8
 C            76.0 2.75  3     67.2     84.8
 P            82.0 1.23  3     78.1     85.9

Results are averaged over the levels of: Block
Confidence level used: 0.95

$contrasts
 contrast estimate   SE df t.ratio p.value
 A - B         6.5 4.26  3   1.525  0.5212
 A - C        -0.5 4.26  3  -0.117  0.9993
 A - P        -6.5 3.01  3  -2.157  0.3110
 B - C        -7.0 4.26  3  -1.642  0.4744
 B - P       -13.0 3.01  3  -4.313  0.0670
 C - P        -6.0 3.01  3  -1.991  0.3564

Results are averaged over the levels of: Block
P value adjustment: tukey method for comparing a
family of 4 estimates
```

The design for the blood pressure monitor experiment is a special case of an incomplete block design that Bechhofer and Tamhane (1981) have called a BTIB (balanced with respect to test treatments). In these designs one treatment level is designated as the control level and there is more interest in comparing each of the other treatment levels with the control than there is in comparing the other treatment levels. In a BTIB design each treatment must appear the same number of times (λ_0) in a block with the control treatment, and each test treatment must occur the same number of times (λ_1) in a block with every other test treatment. This results in a design that is more efficient in comparing each treatment with the control but less efficient in comparisons among the other treatment levels. One way to form a BTIB design with t levels of the treatment factor and block size k is to combine a control level to each block of a BIB design with $t-1$ levels of the treatment factor and block size $k-1$.

The BTIB design, described in the last paragraph, is a special case of a partially balanced incomplete block design (or PBIB) with two associate classes. In these designs each pair of treatments are either first associates or second associates. First associates occur together in a block λ_1 times, and second associates occur together in a block λ_2 times, where $\lambda_1 > \lambda_2$. The standard error of the difference in treatment means for first associates is smaller than the standard error of the difference in means for second associates. There are multiple ways of creating PBIB designs. Bose et al. (1954) have published tables of some of the most useful plans. Jarrett and Hall (1978) have described a class of PBIB designs called generalized cyclic incomplete block designs, which have good statistical properties and are easy to create. Generalized cyclic incomplete block designs with block size k and $b = t$ blocks can be created following the steps listed below.

1. To form a generalized cyclic design with $b = t$,

   (a) Start with a subset of k treatment factor levels as the initial block.

   (b) Add 1 (modulo t) to each treatment level in the initial block to form the next block.

   (c) Continue adding blocks until you have t blocks.

To illustrate this, consider creating an incomplete block design with $t = 6$ and $k = 3$. The BIB design with the smallest number of blocks for this combination is found (solving ***Equations (8.1)*** to ***(8.3)*** to be $b = 10$). A generalized cyclical incomplete block design for testing $t = 6$ levels of the treatment factor can be found with $b = t = 6$ blocks. To find a design with six blocks, following the steps above, start with a subset of $k = 3$ levels of the treatment factor to be tested in the initial block, that is,

$$(1\ 2\ 4)$$

Next add one to each treatment level to get the treatment levels in the next block, that is,

$$(2\ 3\ 5)$$

Continue this $modulo\ 6$ (i.e., $7\ modulo\ 6 = 1$, etc.) to form the following.

Block	Treatments		
1	1	2	4
2	2	3	5
3	3	4	6
4	4	5	1
5	5	6	2
6	6	1	3

The treatment levels in each block would be randomized to the three experimental units in each block. The function `design.cyclic` in the **R** package `agricolae` can create generalized cyclic designs with t blocks of size k. For example, the code below shows the commands to create a generalized cyclic design with $t = 6$, $k = 3$. The first argument for `design.cyclic` is a vector containing the levels of the treatment factor. The second argument is the block size, k, and the third argument is the number of replicates, r, of each treatment level. In this case since there are six blocks of three experimental units there is a total of 18 experimental units and each treatment level will be assigned to three, making $r = 3$.

```
> library(agricolae)
> treat <- c(1, 2, 3, 4, 5, 6)
> des <- design.cyclic(treat, k = 3, r = 3)
```

```
cyclic design
Generator block basic:
1 2 4

Parameters
===================
treatmeans : 6
Block size : 3
Replication: 3
```

The function design.cyclic generates and randomizes the plan. In this example of a PBIB design, each treatment level has one first associate with $\lambda_1 = 2$ and four second associates with $\lambda_2 = 1$. The object created by design.cyclic has two components. The component des$book contains the design as shown below.

```
> des$book
```

```
   plots group block treat
1    101     1     1     1
2    102     1     1     4
3    103     1     1     5
4    104     1     2     3
5    105     1     2     5
6    106     1     2     2
7    107     1     3     5
8    108     1     3     6
9    109     1     3     2
10   110     1     4     6
11   111     1     4     3
12   112     1     4     1
13   113     1     5     4
14   114     1     5     3
15   115     1     5     6
16   116     1     6     1
17   117     1     6     2
18   118     1     6     4
```

The analysis of PBIB designs is the same as the examples shown for BIB designs. The type III sums of squares for treatment and the least squares means should be used. The model and assumptions are the same as for the RCB design, and the assumptions of normality and homogeneity of experimental error variance can be checked with the residual plots.

```
> par( mfrow = c(2,2) )
> plot(modm, which=5)
> plot(modm, which=1)
> plot(modm, which=2)
> plot(residuals(modm) ~ modm$model$Treatment,
main="Residuals vs Exp. Unit", font.main=1,data=bread)
```

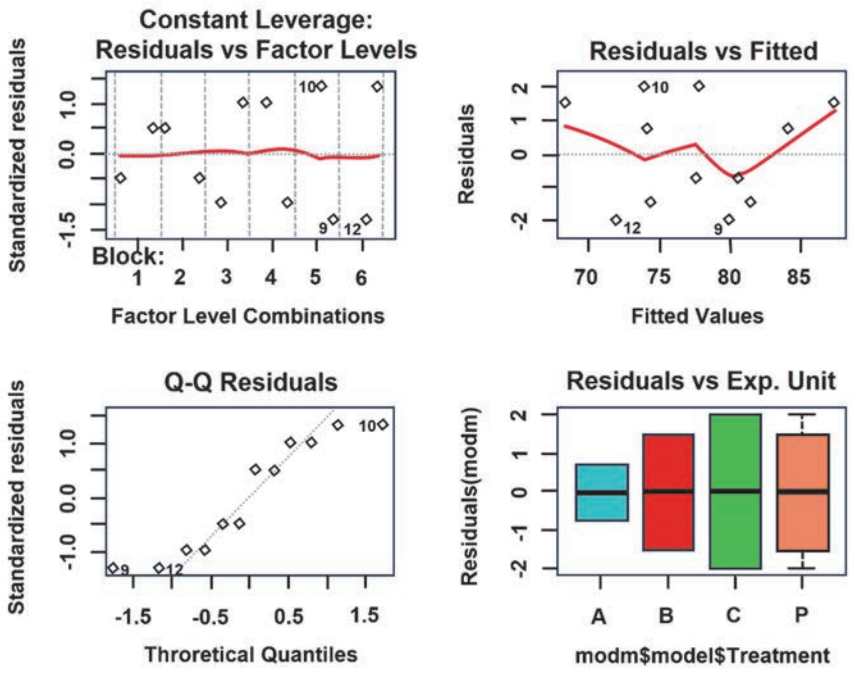


Figure 8-3. Plots to check the normality assumptions.

8.6 Row Column Designs

Latin-square designs with two independent blocking factors were described in Chapter 4. These designs could increase the precision in detecting differences among treatments by adjusting for variability in experimental units in two ways. However, the restriction on Latin-square designs was that the number of levels of the row blocking factor, the number of levels of the column blocking factor, and the number of levels of the treatment factor all had to be equal. This restriction may be impractical in some situations.

Suppose an experiment was conducted where the purpose was to study the effect of shelf facing on the sales of toothpaste in drugstores. In that example, four levels of the treatment factor (shelf facings), four levels of the row blocking factor (stores), and four levels of the column blocking factor (week of sales) were used. If the researchers desired to expand their study to test eight different shelf facings instead of four, they could easily increase the number of levels of the row blocking factor and include eight stores. However, increasing the number of weeks would prolong the study and could be undesirable.

An alternate design called a Row Column design or RCD utilizes a complete block design in the column blocks, but an incomplete block design in the row blocks. This type design can also be created and randomized by the function `design.cyclic` in the **R** package `agricolae`. The code below illustrates how this can be done. The addition of the argument `rowcol=TRUE` causes `design.cyclic` to create a row column design. Since there are eight levels of the treatment factor and column block size $k = 4$, there will be $r = 4$ replicates of each treatment level in the design. The argument `seed=1` fixes the random seed so the same design can be produced in repeat runs.

```
> library(agricolae)
> treat <- c(1, 2, 3, 4, 5, 6, 7, 8)
> RCD <- design.cyclic(treat, k = 4, r = 4, rowcol = TRUE,
seed = 1)
```

```
cyclic design
Generator block basic:
1 2 4 8

Parameters
===================
treatmeans : 8
Block size : 4
Replication: 4
```

This code will create a design with eight row blocks for stores, and four column blocks for weeks. The model and analysis of RCDs is identical to the model and analysis of Latin-square designs described in Chapter 4, with the exception that type III treatment sums of squares and least squares treatment means should be used due to the fact that the row blocks are incomplete and do not contain all levels of the treatment factor.

8.7 Review of Important Concepts

When experimental units are heterogeneous and can be grouped into smaller blocks of more homogeneous experimental units, a blocked design should be used. When the number of experimental units per block or block size is smaller than the number of levels of the treatment factor or combinations of levels of treatment factors in a factorial design, an incomplete block design should be used.

When there is only one treatment factor, there is a choice between two types of incomplete block designs. Balanced incomplete block (BIB) designs require that every treatment level occurs in a block an equal number of times with every other treatment level. BIB designs can be created with the `optBlock` function in the `AlgDesign` package. The advantage of these designs is that the precision (or standard error) of the difference in every possible pair of treatment means will be equal. The disadvantage is that many blocks and experimental units may be required to achieve the balance. The other alternative design for one treatment factor is the partially balanced incomplete block (PBIB) designs.

The advantage of PBIB designs is that the total number of blocks and experimental units required can be reduced. The disadvantage is that the standard error of differences in pairs of treatment means will not be constant. There are many different methods of creating PBIB designs, and some of the more useful designs have been tabled. One type of PBIB called a BTIB (balanced with respect to test treatments) can be easily created from a BIB design. This design is useful when there is more interest in comparing one treatment level (such as a control) to all other treatment levels than there is in comparisons among the other treatment levels. Another class of PBIB designs that can be easily created using the `design.cyclic` function in the `agricolae` package are called generalized cyclic designs.

Latin-square designs introduced in Chapter 4 have two orthogonal blocking factors and contain a complete block design in both the row blocks and column blocks. If an incomplete block design is required in either the row or column blocks, a row column design (RCD) can be utilized.

When experimenting with multiple factors and the block size is not large enough to accommodate all possible treatment combinations, there are two alternative methods for creating an incomplete block design. The first method is to completely confound some interactions with blocks in a completely confounded blocked factorial (CCBF) design, or a completely confounded blocked fractional factorial (CCBFF) design. The advantage of these designs is that the total number of blocks and experimental units can be minimized. The disadvantage is that some interactions will be completely confounded with blocks and will be inestimable. The other method is to use a partially confounded blocked factorial (PCBF) design. ***Figure 8-4*** illustrates when these designs should be used in relation to the designs presented in other chapters.

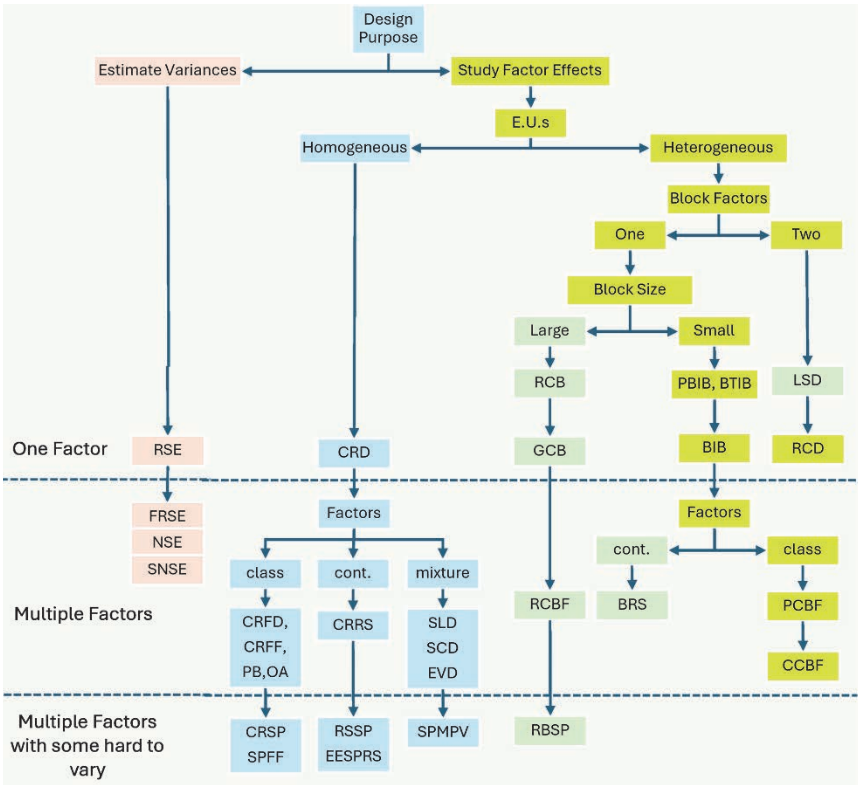


Figure 8-4. Design Selection Roadmap for Software Problem Identification design

8.8 Design Name Acronym Index

RSE	=	random sampling experiment
FRSE	=	factorial random sampling experiment
NSE	=	nested sampling experiment
SNSE	=	staggered nested sampling experiment
CRD	=	completely randomized design
CRFD	=	completely randomized factorial design
CRFF	=	completely randomized fractional factorial
PB	=	Plackett-Burman design
OA	=	orthogonal array design
CRSP	=	completely randomized split plot
RSSP	=	response surface split plot
EESPRS	=	equivalent estimation split-plot response surface
SLD	=	simplex lattice design
SCD	=	simplex centroid design
EVD	=	extreme vertices design
SPMPV	=	split-plot mixture process variable design
RCB	=	randomized complete block
GCB	=	generalized complete block
RCBF	=	randomized complete block factorial
RBSP	=	randomized block split plot
PBIB	=	partially balanced incomplete block
BTIB	=	balanced treatment incomplete block
BIB	=	balance incomplete block
BRS	=	blocked response surface
PCBF	=	partially confounded blocked factorial
CCBF	=	completely confounded blocked factorial
LSD	=	Latin-square design
RCD	=	row-column design

9 CASE STUDY IN COMBINATORIAL TESTING

From (Introducing combinatorial testing in a large organization, 2015)

In 2009, when Lockheed Martin decided to explore the benefits of combinatorial testing methods as a way to reduce testing costs and maintain competitive processes, its decision was based on several developments. (Hagar, Wissink, Kuhn, & Kacker, 2015) That same year, pilot projects using these methods at Eglin Air Force Base in Florida were showing promise in reducing testing costs. The US Department of Defense's (DoD's) Office of Test and Evaluation had recently endorsed the design of experiments (DOE) statistical testing approach, (McQueary, 2009) some principles of which are in combinatorial testing. In the commercial realm, firms were reporting success with pairwise testing, a basic form of combinatorial testing that covers two-way factor combinations.

Together, these developments were sufficient motivation for Lockheed Martin to launch its own pilot projects. The goal was to make combinatorial testing available to its engineers and use it internally without mandates from either customers or management. To aid in this effort, the company developed a cooperative research and development agreement (CRADA) with the National Institute of Standards and Technology (NIST), which is one of NIST's mechanisms for conducting joint research with US industry. CRADAs also provide flexibility in structuring projects, assigning intellectual property rights, and protecting industry- related proprietary information and research results.

Lockheed Martin and NIST entered into the CRADA to better understand the applicability and effectiveness of a relatively new approach for software testing to improve the quality, safety, and reliability of US products and systems. A goal of particular interest was to better understand the challenges in introducing a new software-testing approach in a large US corporation.

Specific goals in introducing the method in Lockheed Martin projects were to

- Evaluate the concept's viability.
- Test process improvement in a variety of domains, including system, software, and hardware testing.
- Make tests more effective in finding problems.
- Reduce testing cost, or at least the test life-cycle cost, by reducing errors found late in development or in the field.

Lockheed Martin entered into this investigation and improvement effort with the assumption that gathering enough sound evidence to introduce real change would take time and a series of efforts. These assumptions stemmed from both its process improvement experience and understanding of what motivates large organizations.

To assess the applicability of combinatorial testing, described in more detail in the sidebar "Understanding Combinatorial Testing," Lockheed Martin evaluated tools, generated sample test cases, and analyzed data from pilot projects. From this process, the company learned many valuable lessons about introducing combinatorial testing into a large corporate body.

9.1 EARLY INVESTIGATIONS

Lockheed Martin's interest in combinational testing actually began in 2005, four years before it launched the pilot projects. At that stage, technical staff reviewed reports on the basic pairwise testing form of combinatorial testing from various industrial sources. The company then contracted with consultants to generate introductory training and informational materials, so that senior staff could get some exposure to the concepts. For experienced testers, Lockheed Martin also held an advanced class on combinatorial testing. However, although testers learned the principles of combinatorial testing, they only occasionally expressed interest in using it.

Once it decided on and funded a pilot evaluation, Lockheed Martin targeted one or two staff members per project to learn combinatorial test concepts and how to use the supporting tools. At this point, R&D had four main goals:

- Conduct evaluation studies to determine which combinatorial test tools might fit company needs.
- Conduct some corporate- funded case studies to evaluate the applicability of combinatorial testing for aspects of ongoing projects.
- Generate an informational website to broaden combinatorial testing knowledge and promote skills development.
- Create a set of online training materials with exercises so that test staff can review and access the classes at any time.

9.1.1 Trade studies

Project staff in large companies are often skeptical about any technology they are not already using, because of the risks associated with using tools without a solid track record. Prototype evaluation studies can minimize the perception of risk by providing a way to assess a new concept without committing to its application. These studies are particularly useful if funding is outside of corporate project funding, as was the case in the combinatorial testing effort.

Lockheed Martin conducted several trade studies, including a comparison to historic F-16 design and test problems (Cunningham, Hagar, & Holman, 2012) and assessments of the method's applicability to visual display and flight testing, the support of vehicle and weapons configuration testing, and digital command system testing. The company selected candidate studies on the basis of test data availability and life- cycle stage.

9.1.2 Preliminary tool evaluation

Recognizing that effective combinatorial testing requires appropriate software tools, the company obtained a series of tools, compared their features, and applied them to real problems. It also created a website to provide a single destination for combinatorial test information and tools. ***Table 9-1*** lists the tools evaluated and their application context.

9.2 PILOT TOOLS AND APPLICATIONS

Each pilot project required supporting tools to implement and support combinatorial testing. For the primary tool, Lockheed Martin chose the Advanced Combinatorial Testing System:

(ACTS: http://csrc.nist.gov/acts)

ACTS is jointly developed by NIST and the University of Texas at Arlington. The company supplemented ACTS as needed with the other tools listed in ***Table 9-1***.

Table 9-1. Tools used in the pilot projects.

Developer	Tool*	Application context
NIST/University of Texas at Arlington	ACTS	Covering array generation; constraint support
Air Academy Associates	SPC XL	Statistical analysis support
	DOE KISS	Support of simple design of experiments analysis
	DOE PRO XL	Design of experiments support
	DFSS Master	General analysis
Phadke & Associates	rdExpert	Historic test data analysis
Hexawise	Hexawise	Covering array generation; Webbased, group collaboration

*ACTS: Advanced Combinatorial Testing System; SPC: Statistical Process Control XL (an Excel plug-in); DOE: Design of Experiments; KISS: Keep It Simple Statistically; DFSS: Design for Six Sigma.

ACTS is a freely downloadable research tool for generating combinatorial t- way test suites based on covering arrays. (Lei Y. , 2008) It has many features that made it ideal for the pilot projects:

- Testers can exclude test- setting combinations that are invalid according to user- specified constraints.

- Two test generation modes support building a test suite from scratch or extending an existing suite to save earlier tests.
- Variable- strength test suites are possible. For example, all factors could be covered with strength 2 and a subset of the factors (known to be interrelated) could be covered with higher strength (Lei Y. , 2008).
- ACTS verifies if the user- supplied test suite covers all t-way combinations.
- The tool comes with a GUI, a command line interface, and an API.

In addition to tools, Lockheed Martin evaluated application areas in systems, software, hardware, and materials testing—areas that its traditional test processes were already addressing but often without a systematic scientific approach. It was interested in applying combinatorial testing to

- Different system configurations, which could have a variety of user, hardware, electronic, and software options.
- Software with a range of parameters and variables that might interact to create errors.
- Configurations in a flight test scenario, which can have many use conditions and hardware configurations, such as different targets, weapon options, altitudes, flight paths, or sensors for targeting.
- Support user interface testing in system software, which can entail many option combinations in menus, user displays, pages, and interactions; and
- test the hardware settings of complex systems, such as switch setting, sensor inputs, signals, and outputs.

Staff quickly suggested other areas for consideration, but the company kept its focus on small initiatives that would help develop a basic understanding of combinatorial testing and tools to support its implementation.

9.3 PILOT PROJECTS AND RESULTS

In all, Lockheed Martin evaluated 14 areas from the candidates in the trade study evaluation. From these, it selected eight of those with the best data availability and strongest management support for its pilot projects.

9.3.1 Flight vehicle mission effectiveness

The project staff created six combinatorial test cases in different mission effectiveness (ME) areas and worked with the existing test team using historic ME plans. The study compared combinatorial testing relative to test cases created from a statistical analysis tool. Results indicated no major difference between the combinatorial testing and statistical approach, but the test team felt that combinatorial tools were easier to use.

9.3.2 Flight vehicle engine failure modes

The project team considered failure combinations, failure accommodation, and fault- tolerance combinations under defined constraints. An analysis of historic plans revealed missed test cases and showed that combinatorial testing would yield 30 percent better coverage.

9.3.3 Flight vehicle engine upgrade flight and lab testing

The team generated six combinatorial test cases covering vehicle variations in various flight modes and conditions. The generated cases were statistically equal to existing team test plans.

9.3.4 F- 16 ventral fin redesign flight test program

The F-16 ventral fin redesign effort (Cunningham, Hagar, & Holman, 2012) is a case study that demonstrates how combinatorial testing can be applied to design analysis problems, solutions, and evaluation. The team generated a set of combinatorial test studies, data sets, and reports that recreated the original analysis but with combinatorial testing instead of highly experienced expert judgment. The team found that combinatorial testing would have reduced the number of tests by roughly 20 percent.

9.3.5 Electronic warfare system testing

The team generated four combinatorial test cases from existing test plans. The idea was to show the validity of using a combinatorial test tool to analyze word-based test plans. Results demonstrated the feasibility of the *rdExpert* tool's features to analyze existing test plans while providing improvements by reducing the total number of test cases by 10 percent.

9.3.6 System test flight parameters

Parameters included navigation accuracy, electronic warfare performance, sensor information, and radar detection performance test points within system test flights.

Combinatorial testing revealed the possibility of generating effective test cases with diverse subsystem interactions.

9.3.7 Electromagnetic effects engineering

The team analyzed the existing test plan and generated new tests. They saw no coverage improvement with combinatorial testing, but they felt that the tool might have generated cases faster than was possible with the human analysis and judgment that was part of the existing test plan.

9.3.8 Digital command system testing

The system had 16 commands for performing various file operations in three file systems. Most commands had more than one parameter, and some parameters accepted up to three values. Also, some parameters accepted a continuous range of integer values, so the project team added constraints to limit impossible operations.

The team produced 26 test cases spanning 1,148 test combinations. The tests uncovered several major bugs that had gone undetected through unit testing.

9.4 EVALUATION OF PILOT PROJECTS

Beyond using only classic requirements-based testing, the combinatorial testing efforts demonstrated that the methodology had the potential for effective and affordable tests in software, hardware, system, and flight testing. Combinatorial testing was applicable when the test structure involved multiple options or parameters.

9.4.1 Culture change

Combinatorial testing requires some culture change for systems engineering and test teams. The pilot effort showed that organizations must incorporate the method into early test planning as well as provide training and management support. Lockheed Martin's initial estimate is that combinatorial testing and supporting technology can save up to 20 percent in test planning and design costs if used early on in a program and can increase test coverage by 20 to 50 percent.

The tool investigation and pilot studies were generally positive, and Lockheed Martin has already identified possible future efforts. All the combinatorial test tools investigated worked, and results were similar. Each tool had features that would make it viable given certain criteria. For example, if cost was the main issue, open-source tools would be the best choice. Some tools had better test-plan analysis features; yet others would work better with distributed teams. Overall Lockheed Martin's tool studies revealed that combinatorial test tools are maturing nicely.

9.4.2 General observations

Staff from Lockheed Martin reviewed project problems and results and documented general observations as positive results or mixed.

Positive results came mainly from recognition of the problems that combinatorial testing revealed. An example is the F-16 project team's discovery that staff cost could have been lower. Many project staff members were generally impressed with the method's ability to enhance and supplement existing test knowledge and abilities.

After the pilot projects ended, several project teams continued using the ACTS tool and found software errors before product release. These positive results became data points for engineers who wanted more evidence of combinatorial testing's viability.

Mixed results centered on attitudes about cost versus improvement gains. Several projects that analyzed existing test plans did find weaknesses in testcase coverage but did not want to add tests because of the resulting increased test cost. After applying combinatorial testing, another project found at least one new error, but felt that combinatorial testing was not a vast improvement. However, they had no similar project for comparison.

9.4.3 Lessons learned

Testing teams tend to have fairly rigid processes, and introducing new ideas to change the testing mindset can prove difficult unless a champion takes on the task of promulgating combinatorial testing.

Because their expectations about improvement were sometimes unrealistic, teams often found it hard to see where and how to apply combinatorial testing to existing test efforts. Some teams found the number of tests had increased, and they had expected combinatorial testing to reduce that number. They failed to recognize that the original test set had to increase because it did not provide good coverage in the first place. In addition, the teams did not fully appreciate how combinatorial methods could help identify weaknesses in test sets that are intended to be thorough but are not.

Lockheed Martin also learned lessons about the role of training in achieving method buy- in. Teams that expressed interest in using combinatorial testing failed to adopt it because of cost, skill, or time.

The company now offers training and tools to help compensate for the lack of follow- through in some of these areas.

Lack of training also contributes to resistance in using tools to support new methods. Because many testers were not trained in combinatorial testing, they saw the supporting tools as foreign to their experience and thus resisted using them. Lockheed Martin now provides online training and familiarization to start overcoming this resistance.

Training is essential for internal technology transfer as well. The company established a website for engineers featuring a basic introduction to combinatorial testing and the Design of Experiments method, webinar materials, and white papers, as well as links to tools, external websites, training, and presentation packages. The company also enhanced online training materials with the knowledge gained from R&D efforts. Staff built small training modules as prototype classes with exercises on combinatorial testing and tool use and made them Web accessible.

9.5 INCREASING ACCEPTANCE

Results from the pilot projects suggest ways to broaden the acceptance of combinatorial testing. Approaches to integrating this method and supporting tools into existing practice must provide support materials that include training packages and data on cost and effectiveness. Tools must be easy to use and integrate readily with industry test infrastructures and frameworks, such as allowing an automatic import into model- based test tools.

Perhaps, most important, organizations need to adapt combinatorial testing and supporting tools to fit their existing procedures.

9.5.1 Improved support materials and guidance

To improve combinatorial testing support, Lockheed Martin is continuing to develop training support, including textbooks and classes, case studies, and industry sites with additional information. Further, the addition of combinatorial testing to standards such as ISO/IEC/IEEE

29119 or DoD standards will likely encourage the adoption of math-based methods into software testing from the beginning of a project's life cycle.

Internally, organizations must provide rigorous hands- on training. Lockheed Martin plans to enhance its online forum and provide more such training for staff. It also plans to encourage subject-matter experts to promote combinatorial testing on projects; educate management and highlight projects that are already using combinatorial testing. In documenting our pilot project experience, we hope to illustrate how the measurement and evaluation of pilot projects can advance software engineering through the increased adoption of scientific test methods.

9.5.2 Adapting to existing procedures

It is not always practical to redesign an organization's testing procedures to use tests based on covering arrays. Testing procedures often develop over time, and employees have extensive experience with a particular approach. Units of the organization may be structured around established, documented test procedures, particularly in organizations that must test according to contractual requirements or standards.

Because much software assurance involves testing applications that have been modified to meet new specifications, the organization is likely to have an extensive library of legacy tests—tests that it can reuse to save time and money. These tests might not be based on covering arrays.

If creating new test suites is not an option, the organization can glean the advantages of combinatorial testing by measuring the combinatorial coverage of existing tests and then supplementing those with additional tests as needed. Building covering arrays for some specified level of t is one way to provide t-way coverage. However, many large test suites naturally cover a high percentage of t-way combinations. For example, tests developed for NASA spacecraft provided better than 90 percent pairwise coverage despite being developed without combinatorial testing tools. (Maximoff, Kuhn, Trela, & Kacker, 2010) If an existing test suite covers almost all t- way combinations for an appropriate level of t, the suite might be sufficient for the required assurance.

Determining the level of input or configuration-space coverage can also help in understanding the degree of risk that remains after testing. If testing has covered 90 to 100 percent of the relevant state space, the risk is likely to be smaller than it would be after testing that covers a much smaller portion of that space. These considerations led NIST to develop an approach that measures an existing test suite's combinatorial coverage and then automatically extends it to provide the desired coverage level. NIST has developed a sophisticated tool based on this approach, which also allows for constraints among variables. (Kuhn, Dominguez, Kacker, & Lei, 2013)

Early efforts to include combinatorial testing into the Lockheed Martin test culture have been largely positive. Projects within the company have taken advantage of the tools, website, and training. Individual engineers have been supportive and even enthusiastic. Traffic on the combinatorial testing website has grown, and interest in improving the prototype online training materials continues.

Existing projects have been somewhat slow to consider combinatorial testing, most likely because project staff are risk averse and thus wary of new ideas that are not mandated under a contract. Lockheed Martin expects that new contracts and improvement efforts will be more likely to use combinatorial testing.

The company supports such improvement efforts and plans to continue using the method, related test tools, training and websites, and management support. The work we have described reflects early efforts, and Lockheed Martin takes a long- term view of improvement. Engineers look forward to working with industry, government, tool vendors, researchers, and others to promote combinatorial testing. NIST is already incorporating lessons from this cooperative research effort into generic advanced measurement and testing research to promote improved quality, safety, and reliability in software and systems.

APPENDIX A. EMPIRICAL DATA ON SOFTWARE FAILURES

One of the most important questions in software testing is "how much is enough"? For combinatorial testing, this question includes determining the appropriate level of interaction that should be tested. That is, if some failure is triggered only by an unusual combination of more than two values, how many testing combinations are enough to detect all errors? What degree of interaction occurs in real system failures? This section summarizes what is known about these questions based on research by NIST and others [4, 7, 34, 35, 36, 65].

Table A-1 below summarizes what we know from empirical studies of a variety of application domains, showing the percentage of failures that are triggered by the interaction of one to six variables. For example, 66% of the medical devices were triggered by a single variable value, and 97% were triggered by either one or two variables interacting. Although certainly not conclusive, the available data suggest that the number of interactions involved in system failures is relatively low, with a maximum from 4 to 6 in the six studies cited below. (Note: TCAS study used seeded errors, all others are "naturally occurring", * = not reported.)

Table A-1. Number of variables involved in triggering software failures

Vars	Medical Devices	Browser	Server	NASA GSFC	Network Security	TCAS
1	66	29	42	68	17	*
2	97	76	70	93	62	**53**
3	99	95	89	98	87	**74**
4	100	97	96	100	98	**89**
5		99	96		100	**100**
6		**100**	**100**			

Table A-2. System characteristics

System	System type	Release stage	Size (LOC)
Medical Devices	Embedded	Fielded products	**$10^3 – 10^4$ (varies)**
Browser	Web browser	Development/ beta release	**approx. 2 x 10^5**
Server	HTTP server	Development/ beta release	**approx. 10^5**
NASA database	Distributed scientific database	Development, integration test	**approx. 10^5**
Network security	**Network protocols**	**Fielded products**	**$10^3 – 10^5$ (varies)**

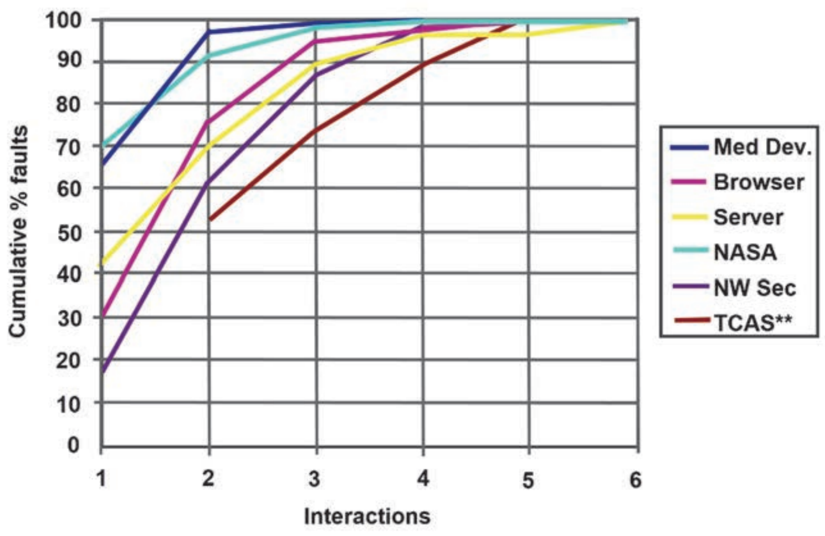


Figure A-1. Cumulative percentage of failures triggered by t-way interactions.

We have also investigated a particular class of vulnerabilities, denial-of-service, using reports from the National Vulnerability Database (NVD), a publicly available repository of data on all publicly reported software security vulnerabilities. NVD can be queried for fine-granularity reports on vulnerabilities. Data from 3,045 denial-of- service vulnerabilities have the distribution shown in ***Table A-3***. We present this data separately from that above because it covers only one particular kind of failure, rather than data on any failures occurring in a particular program as shown in ***Figure A-1***.

Table A-3. Cumulative percentage of denial-of-service vulnerabilities triggered by t-way interactions.

Vars	NVD cumulative %
1	**93%**
2	**99%**
3	**100%**
4	**100%**
5	**100%**
6	**100%**

Why do the failure detection curves look this way? That is, why does the error rate tail off so rapidly with more variables interacting? One possibility is that there are simply few complex interactions in branching points in software. If few branches involve 4-way, 5-way, or 6-way interactions among variables, then this degree of interaction could be rare for failures as well. The table below (***Table A-4*** and ***Figure A-2***) gives the number and percentage of branches in avionics code triggered by one to 19 variables. This distribution was developed by analyzing data in a report on the use of MCDC testing in avionics software [16], which contains 20,256 logic expressions in five different airborne systems in two different airplane models. The table below includes all 7,685 expressions from *if* and *while* statements; expressions from assignment (:=) statements were excluded.

Table A-4. Number of variables in avionics software branches

Vars	Count	Pct	Cumulative
1	5691	74.1%	**74.1%**
2	1509	19.6%	**93.7%**
3	344	4.5%	**98.2%**
4	91	1.2%	**99.3%**
5	23	0.3%	**99.6%**

6	8	0.1%	**99.8%**
7	6	0.1%	**99.8%**
8	8	0.1%	**99.9%**
9	3	0.0%	**100.0%**
15	1	0.0%	**100.0%**
19	**1**	**0.0%**	**100.0%**

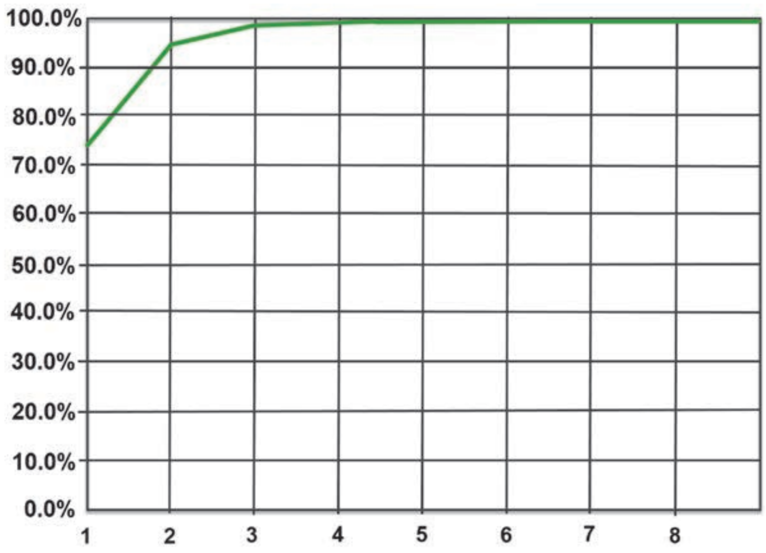


Figure A-2. Cumulative percentage of branches containing n variables.

As shown in ***Figure A-2***, most branching statement expressions are simple, with over 70% containing only a single variable. Superimposing the curve from ***Figure A-2*** on ***Figure A-1***, we see (***Figure A-3***) that most failures are triggered by more complex interactions among variables. It is interesting that the NASA distributed database failures, from development-phase software bug reports, have a distribution similar to expressions in branching statements. This distribution may be because this was development-phase rather than fielded software like all other types reported in ***Figure A-1***. As failures are removed, the remaining failures may be harder to find because they require the interaction of more variables. Thus testing and use may push the curve down and to the right.

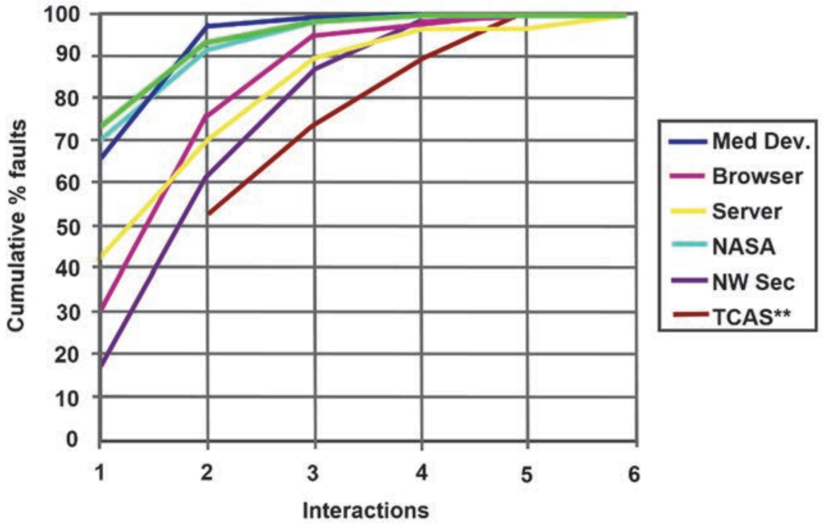


Figure A-3. Branch distribution (green) superimposed on Figure 1.

APPENDIX B. OTHER AVAILABLE TOOLS

I updated this table on 18 April 2024. About half of my previous list of tools no longer exist or are not being maintained.

Tool	Authors	Notes
AllPairs	Satisfice	Perl script, free, GPL
Pro-Test	SigmaZone	GUI, commercial, https://sigmazone.com/protest-tutorial/
Jenny	[Jenkins]	Command-line, free, public-domain, http://burtleburtle.net/bob/math/jenny.html
Test Vector Generator		GUI, free, http://sourceforge.net/projects/tvg/
PICT	Microsoft Corp.	Command-line, open source at http://github.com/microsoft/pict
OATSGen	Motorola	
Rocket® SmartTest ™	Smartware Technologies Inc.	GUI, commercial, https://www.rocketsoftware.com/products/rocket-mainframe-application-testing-and-debugging/rocket-smarttest
AllPairs	MetaCommunications	Free, https://www.satisfice.com/download/allpairs
ACTS	NIST	GUI, https://csrc.nist.gov/projects/automated-combinatorial-testing-for-software
Bender RBT Inc.	BenderRBT	GUI, commercial, http://www.benderrbt.com/bendersoftware.htm
Pairwise Pict Online	TestersDesk	Free, https://pairwise.yuuniworks.com/
Pairwise Testing Generator		Command-line, free, https://mecsimcalc.com/app/8809236/pairwise_testing_generator
VPTAG	[Robert Vanderwall]	http://sourceforge.net/projects/vptag/
Hexawise	Hexawise	Web-based, free & commercial, https://hexawise.com/
NTestCaseBuilder	[Murphy]	.NET library, https://www.nuget.org/packages/NTestCaseBuilder/
tcases	[Kimbrough]	Command-line, Maven plugin, free, https://github.com/Cornutum/tcases
Pairwiser	Inductive AS	Web-based, free & commercial, https://inductive.no/pairwiser/
NUnit	Poole et al	Unit test framework, https://docs.nunit.org/
ecFeed	ecFeed AS	Standalone, Eclipse plug-in, and jUnit runner, https://ecfeed.com/

Tool	Authors	Notes
Pairwise Online Tool	[Dementiev]	Web-based, free, https://pairwise.teremokgames.com/4s8/
JCUnit	[Ukai]	Unit test framework, https://github.com/dakusui/jcunit
CAGen	SBA Research	Web-based and command-line, https://srd.sba-research.org/tools/cagen/#/workspaces
CTWedge	University of Bergamo	Web-based, https://foselab.unibg.it/ctwedge/
CAMetrics	SBA Research	https://matris.sba-research.org/cametrics/
AllPairsPy	[Hombashi]	Python library, https://github.com/thombashi/allpairspy/
CoverTable	[Yasuyuki]	Python and TypeScript. Open source, https://github.com/walkframe/covertable
UnitTestDesign	[Dolgert]	Julia library, https://github.com/adolgert/UnitTestDesign.jl
Kiwi TCMS	Open source community	Web-based. Supports manual testing. https://kiwitcms.org/
AllPairs4J	[Pavelicii]	Java library, https://github.com/pavelicii/allpairs4j

References

Ammann, P., & Offutt, J. (2008). *Introduction to Software Testing, .* New York: Cambridge University Press.

Ammann, P., & Offutt, J. (2016). *Introduction to software testing. .* Cambridge University Press.

Bechhofer, R. E., & Tamhane, A. C. (1981). Incomplete block designs for. *Technometrics, 23*, 45-57.

Beizer, B. (2003). *Software testing techniques.* Dreamtech Press.

Boehm, B. W. (1981). *Software Engineering Economics.* Prentice Hall.

Bose, R. C., Clatworthy, W. H., & Shrikhande, S. S. (1954). *Tables of partially balanced designs with two associate classes. Technical Bulletin 107.* North Carolina Agricultural Experiment Station.

Bousquet, L. d., Ledru, Y., Maury, O., Oriat, C., & Lanet, J. (2004). A case study in JML-based software validation. *Proceedings of 19th Int. IEEE Conf. on Automated Sofware Engineering* (pp. 294-297). Linz: IEEE.

Bryce, R., Colbourn, C. J., & Cohen, M. B. (2005). A Framework of Greedy Methods for Constructing Interaction Tests. *Proc. IEEE CS 27th Int'l Conf. Software Engineering (ICSE 05)*, 146–155.

Cohen, M. B., Colbourn, C. J., & Ling, A. (2003). Constructing Strength Three Covering Arrays with Augmented Annealing. *Discrete Mathematics, 308*(13), 2709–2722.

Colbourn, C. (2022, June 26). *Covering Array Tables for t=2,3,4,5,6*. (Arizona State University in Tempe, Arizona) Retrieved from Charlie Colbourn, Professor of Computer Science and Engineering in the School of Computing, Informatics and Decision Systems Engineering: https://www.public.asu.edu/~ccolbou/src/tabby/catable.html

Colbourn, C. J. (2004). Combinatorial Aspects of Covering Arrays. *Le Matematiche (Catania), 58*, 121–167.

Copeland, L. (2004). *A Practitioner's Guide to Software Test Design.* Boston: Artech House Publishers.

Cunningham, A. M., Hagar, J., & Holman, R. J. (2012). A System Analysis Study Comparing Reverse Engineered Combinatorial Testing to Expert Judgment. *Proc. 5th IEEE Int'l Conf. Software Testing, Verification and Validation (ICST 12)*, (pp. 630–635).

Ferrell, T. K., & Ferrell, U. D. (2017). *RTCA DO-178C/EUROCAE ED-12C.* Digital Avionics.

Fisher, R. A. (1926). The arrangement of filed experiments. *Journal of the Ministry of Agriculture of Great Britain,*, 503-513.

Fisher, R. A. (1935). *The Design of Experiments.* Edinburgh,: Oliver and Boyd.

Fisher, R. A. (1940). An examination of possible different solutions of a problem in incomplete blocks. *Annals of Eugenics, 9*, 353-400.

Grochtmann, M., & Grimm, K. (1993). Classification trees for partition testing. *Software Testing, Verification and Reliability, 3*(2), 63-82.

Hagar, J. D., Wissink, T. L., Kuhn, D. R., & Kacker, R. N. (2015, April). Introducing combinatorial testing in a large organization. *Computer (IEEE), 48*(4), 64-72. doi:http://dx.doi.org/10.1109/MC.2015.114

Hartman, A. (2006). Software and Hardware Testing Using Combinatorial Covering Suites. *Operations Research/ Computer Science Interfaces Series, 34*(3), 237-266. doi:DOI: 10.1007/0-387-25036-0_10

Hedayat, A. S., Stufken, J., & Sloane, N. (1999). *Orthogonal Arrays: Theory and Applications.* Springer Science & Business Media.

Hinkelmann, K., & Kempthorne, O. (1994). *Design and Analysis of Experiments* (Vol. I). New York: John Wiley & Sons,.

Hunter, J. S. (1983). Let's all beware the latin square. *Quality Engineering, 1*, 195{198.

Izquierdo-Marquez, I., Torres-Jimenez, J., Acevedo-Juárez, B., & Avila-George, H. (2018). A greedy-metaheuristic 3-stage approach to construct covering arrays. *Information Sciences*, 172-189. doi:https://doi.org/10.1016/j.ins.2018.05.047

Jarrett, R. G., & Hall, W. B. (1978). Generalized cyclic incomplete block designs. *Biometrika, 65*, 397-401.

Kacker, R. N. (1985). Off- Line Quality Control, Parameter Design, and the Taguchi Method. *Journal of Quality Technology, 17*, 176–209.

Katona, G. (1973). Two applications (for search theory and truth functions) of Sperner type theorems. *Periodica Mathematica, 3*, 19–26.

Kiefer, J. (1958). On the nonrandomized optimality and randomized nonoptimality of symmetrical designs. *Annals of Mathematical Statistics*, 675-699.

Kleitman, D. J., & Spencer, J. (1973). Families of k-independent sets. *Discrete Math, 6*, 255-262.

Kshirsager, A. M. (1938). A note on incomplete block designs. *Annals of Mathematical Statistics, 29*, 907-910.

Kuhn, D. R. (2023). Assured Autonomy through Combinatorial Methods. *6th IEEE Dependable and Secure Computing conference.* Tampa, FL: University of South Florida. Retrieved from https://csrc.nist.gov/csrc/media/Projects/automated-combinatorial-testing-for-software/documents/IEEE-DSC-23.pdf

Kuhn, D. R., & Okun, V. (2006). Pseudo-exhaustive Testing for Software. *Proceedings of 30th NASA/IEEE Software Engineering Workshop*, (pp. 153-158).

Kuhn, D. R., Dominguez, I., Kacker, R., & Lei, Y. (2013). Combinatorial Coverage Measurement Concepts and Application. *2nd Intl Workshop on Combinatorial Testing.* Luxembourg: IEEE.

Kuhn, D. R., Kacker, R. N., & Lei, Y. (2010, October). Practical Combinatorial Testing. *National Institute of Standards and Technology (NIST) Special Publication (SP) 800-142*, p. 82. doi:http://dx.doi.org/10.6028/NIST.SP.800-142

Kuhn, D. R., Wallace, D. R., & Gallo, A. J. (2004, June). Software Fault Interactions and Implications for Software Testing. *IEEE Trans. on Software Engineering, 30*(6).

Kuhn, D., Kacker, R. N., & Lei, Y. (2013). *Introduction to Combinatorial Testing.* CRC Press.

Kuhn, R. D., Kacker, R. N., Lei, Y., & Simos, D. (2020, January). Input Space Coverage Matters. *IEEE Computer, 53*(1).

Kuhn, R., Kacker, R. L., & J. Hunter, J. (2009). Combinatorial Software Testing. *IEEE Computer, 42*(8).

Kuhn, R., Kacker, R., Feldman, L., & Witte, G. (2016, May). Combinatorial Testing For Cybersecurity and Reliability. *ITL Bulletin for May 2016*.

Lei, Y. (2008). IPOG/IPOG- D: Efficient Test Generation for Multi- Way Combinatorial Testing. *Software Testing, Verification, and Reliability, 18*(3), 125–148.

Lei, Y., & Tai, K. C. (1998). In- Parameter- Order: A Test Generation Strategy for Pairwise Testing. *Proc. 3rd IEEE Int'l Symp. High-Assurance Systems Eng.*, 254–261.

Lekivetz, R., & Morgan, J. (2020). Covering arrays: using prior information for construction, evaluation and to facilitate fault localization. *Journal of Statistical Theory and Practice, 17*(7). doi:https://doi.org/10.1007/s42519-019-0075-2

Lyu, M. e. (1996). *Software Reliability Engineering.* New York: McGraw Hill.

Maximoff, J. R., Kuhn, D. R., Trela, M. D., & Kacker, R. (2010). A method for analyzing system state-space coverage within a t-wise testing framework. *In 2010 IEEE International Systems Conference* (pp. 598-603). IEEE.

McQueary, C. (2009). *Using Design of Experiments for Operational Test and Evaluation.* Office of the Secretary of Defense. Washington, D.C.: Office of the Secretary of Defense. Retrieved from www.dote.osd .mil/pub/policies/2009/200905Using DoEforOTE_MOA.pdf

Meyer, B. (1997). *Object-Oriented Software Construction, Second Edition.* Prentice Hall.

Montgomery, D. C. (2019). *Design and Analysis of Experiments* (10th ed.). New York: John Wiley & Sons, Inc.

Moskowitz, H. (1988). *Applied Sensory Analysis of Foods.* Boca Raton, FL: CRC Press.

Ostrand, T. J., & Balcer, M. J. (1988). The category-partition method for specifying and generating fuctional tests. *Communications of the ACM, 31*(6), 676-686.

Pérez-Espinosa, H., Avila-George, H., Rodríguez-Jacobo, J., Mendoza, H. A., Martínez-Miranda, J., & Curiel, I. E. (2016). Tuning the Parameters of a Convolutional Artificial Neural Network by Using Covering Arrays. *Research in Computer Science, 1*(21), 69-81. doi:10.13053/rcs-121-1-6

Phadke, M. S. (1989). *Quality Engineering Using Robust Design.* Prentice Hall.

RTCA. (1992). *RTCA/DO-178B - Software Considerations in Airborne Systems and Equipment Certification.* Radio Technical Commission of Aeronautics Inc. (RTCA).

Shasha, D. E., Kouranov, A. Y., Lejay, L. V., Coruzzi, G. M., & Chou, M. F. (2001). Using combinatorial design to study regulation by multiple input signals: a tool for parsimony in the post-genomics era. *Plant Physiology, 127*(4), 1590–1594. doi:https://doi.org/10.1104/pp.010683

Sloane, N. (1993). Covering Arrays and Intersecting Codes. *Journal of Combinatorial Designs, 1*(1), 51–63.

Smith, R., Jarman, D., Bellows, J., Kuhn, R., Kacker, R., & Simos, D. (2019a). Measuring Combinatorial Coverage at Adobe. *Presented at IEEE International Conference on Software Testing, Verification and Validation.* Xian, China: IEEE Xplore.

Smith, R., Jarman, D., Kuhn, R., Kacker, R., Simos, D., Kampel, L., . . . Gosney, G. (2019b). Applying Combinatorial Testing to Large-scale Data Processing at Adobe. *Presented at IEEE International*

Conference on Software Testing, Verification and Validation. Xian, China: IEEE Xplore.

Taguchi, G. (1987). *System of Experimental Design: Volumes 1 and 2.* Kraus Int'l.

Wissink, T., & Amaro, C. (2006). Successful test automation for software maintenance. *2006 22nd IEEE International Conference on Software Maintenance*, (pp. 265– 266).