INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Université d'Ottawa · University of Ottawa

SOFTWARE COMPONENT INTERACTION TESTING:

COVERAGE MEASUREMENT AND GENERATION OF CONFIGURATIONS

by

Alan Webber Williams

A thesis submitted to the Faculty of Graduate and Post-Doctoral Studies in partial fulfilment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

Ottawa-Carleton Institute for Computer Science,
School of Information Technology and Engineering,
University of Ottawa,
Ottawa ON Canada

© 2002 Alan W. Williams

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-68012-6

Canada

Abstract

Systems constructed from components, including distributed systems, consist of a number of elements that interact with each other. When a system is integrated, there may be undesired interactions among those components that cause system failures.

There are two complementary problems in testing a software system. The first problem is to create a test suite, given a description of the expected behaviour of a system configuration. The second problem is to deal with a large number of distinct test configurations. We investigate the second problem in this thesis: the situation when there are various system parameters, each of which can take on a value from a discrete set.

The trade-off that the system tester faces is the thoroughness of test configuration coverage, versus availability of limited resources. We introduce a coverage measure that can provide a basis for determining a set of configurations with "sufficient" coverage, or for evaluation of a set of test configurations that already exists.

This thesis addresses the problem of testing interactions among components of a software system: the "interaction test coverage" problem. We formally define this problem, and give it a set-theoretic framework. This is done through the introduction of an "interaction element," which becomes the unit of test coverage. The problem is compared to, and distinguished from, the minimum set cover problem and the {0,1} integer programming problem. As a result, the status of the NP-completeness of this problem remains open.

Methods from statistical experimental design are introduced, and applied to the problem of generating a set of configurations that achieve coverage of all pair-wise combinations of parameter values. We present a fast, deterministic algorithm to generate such a set of test configurations. The method is compared with other methods, and shown to produce fewer configurations in most situations. The number of configurations generated is logarithmic in the number of parameters, and polynomial in the number of values per parameter. As a result, the number of configurations is usually feasible in practice, and is a significant reduction from the number of possible configurations.

Acknowledgements

First, I would like to thank my supervisor, Dr. Robert L. Probert, for his technical, administrative, and financial support for this work. I must also credit him for the opportunities he has arranged to collaborate with industry for this work, and for other research. His help has made taking this degree a truly positive, and worth while experience.

I would also like to thank the members of my Ph.D. committee, Dr. Hasan Ural and Dr. Lucia Moura of the University of Ottawa, Dr. Louis Nel of Carleton University, and Dr. Lee White of Case Western Reserve University for their suggestions on how to improve this work. Their insights have led to some results that otherwise might have been missed. Dr. R.J.A. Buhr of Carleton University is also acknowledged for his contributions during the early stages of this work.

Thanks also to Dr. Brett Stevens, of Carleton University, for discussions which generated a host of ideas for this work, and for potential future collaboration.

I would also like to thank the members of the Telecommunications Software Research Group during this research. Extra special thanks are due to Louise Desrochers, for her administrative assistance, and without whom chaos would result.

Members of the group who have contributed feedback and ideas include Dr. Gregor v. Bochmann, Dr. Luigi Logrippo, and graduate students Ning Lew, Halim ben Hajla, Anandra Bongari, Daniel Amyot, Rossana Andrade, Khaled el-Fakih, Igor Sales, Rodd Lamarche, Hans van der Schoot, Keqin Zhu, Khalid Khidhir, Nicolas Gorse, Leila Charfi, Ally Li, Jun Li, David Whittier, and Victor Sawma. Thanks also for feedback from fellow researchers who attended seminars about this work, including Jelber Sayyad, Viviana Nastase, and Fernanda Caropreso.

I would also like to credit some patient audiences who saw various early presentations of this material. It is one thing to come up with an idea, and another thing entirely to be

able to explain the idea in an understandable way. Two of the best audience groups have been the attendees at the annual University of Ottawa Graduate Student Association inter-disciplinary conferences, and the students in the three sections of the CSI 4118 "Computer Networks and Protocols" course that I have taught during the term of my studies. Both of these groups have asked insightful questions, and have brought fresh and varied views to this work.

Thanks are also due to the people at the Centre for Advanced Studies at IBM Canada in Toronto for their hospitality during the visit to their lab for the summer 1999 term. The interaction with other Ph.D. students from across Canada during the visit was especially valuable.

I would also like to thank people at ObjecTime Ltd. (now part of Rational Corp.), and at Mitel Corporation, for the opportunity to work on some interesting and fruitful academic-industrial collaborations. Special thanks to Bran Selic at ObjectTime, and at Mitel, Richard Plackowski, Gabriela Alexiu, Steve Slater, Peter Perry, and Daisy Fung.

Finally, I would also like to thank my family for their support, encouragement, advice, and occasional goading: parents Choné and Kenneth Williams; sisters Dr. Carol Hulls and Cynthia Smith, and brothers-in-law Michael Hulls and Dale Smith. Thanks also to my niece and nephew, Maggie Hulls and Carey Hulls, for reminding me of the joy of learning.

The author gratefully acknowledges the support of: the IBM Centre for Advanced Studies, Nortel Networks Ltd., Mitel Corporation, ObjecTime Ltd., Communications and Information Technology Ontario (CITO), the Natural Sciences and Engineering Research Council of Canada (NSERC), and the University of Ottawa.

Table of Contents

Abstract 2
Acknowledgements 3
Table of Contents 5
Index of Figures 9
Index of Tables 11
Index of Definitions 12
Index of Algorithms 14
Index of Lemmas and Theorems 15
Glossary of Variables 16
Chapter 1 Introduction 18
1.1 Contributions of this thesis 20
1.2 Notation conventions 20
Chapter 2 Software Component Interaction Testing in Practice 22
2.1 Chapter Introduction 22
2.2 Substitutable Components 23
2.2.1 Network Component Interaction Testing 23
2.2.2 Commercial off-the-shelf (COTS) components 25
2.3 Object-oriented and Use-case Based Testing 27
2.3.1 Testing in the presence of polymorphism and inheritance 28
2.3.2 Testing Java interfaces 29
2.3.3 Real-time Object-Oriented Methodology (ROOM) substitutable actors 31
2.3.4 Stubs in Use Case Maps (UCMs) 32
2.4 Test data selection 33
2.4.1 Equivalence class partition testing 33
2.4.2 User configurable tool parameters 35
2.5 A reasonable approach: coverage of pair-wise interactions 36
2.6 Previous work on pair-wise (or *d*-way) interactions 37
2.7 Chapter summary 41
Chapter 3 The Interaction Test Coverage Problem 42
3.1 Chapter introduction 42

3.2 Selecting test configurations to cover parameter value interactions 42
3.2.1 Selection of test configurations 43
3.2.2 Interaction Elements 46
3.2.3 The Interaction Test Coverage Problem 51
3.3 Measurement of Interaction Test Coverage 56
3.3.1 Definition 56
3.3.2 Example of Coverage Measurement 57
3.4 Comparison with well-known combinatorial optimization problems 59
3.4.1 Comparison with the Minimum Set Cover problem 60
3.4.2 Formulation as a constraint-based problem 61
3.5 Chapter Summary 67
Chapter 4 Statistical Experimental Design 69
4.1 Chapter Introduction 69
4.2 Principles of experimental design 69
4.3 Orthogonal arrays 70
4.3.1 Properties of Orthogonal arrays 73
4.4 Orthogonal array construction 75
4.4.1 Latin squares 75
4.4.2 Orthogonal Latin squares 76
4.4.3 Algorithm for construction of orthogonal Latin squares 78
4.4.4 Example: Construct three orthogonal Latin squares of order four 81
4.4.5 Orthogonal array construction 83
4.5 Relevant previous work related to orthogonal arrays 89
4.6 Chapter summary 90
Chapter 5 Applying experimental design to software system testing 92
5.1 Chapter introduction 92
5.2 Applying the principles of experimental design to software system testing 92
5.3 Definition of covering array 95
5.4 Previous work 100
5.4.1 Previous work on covering array theory applied to testing 100
5.4.2 Previous work on covering array tools 102
5.5 Chapter summary 106
Chapter 6 An algorithm for covering array construction 107

6.1 Chapter introduction 107
6.2 Covering array construction 107
6.2.1 Building blocks 108
6.2.2 Recursive covering array construction 111
6.2.3 Additional asymmetric columns 116
6.3 Building the covering array 119
6.4 Proof that the algorithm results in a covering array 126
6.5 Number of configurations generated by the algorithm 146
6.6 Complexity of the algorithm 148
6.7 Chapter summary 152
Chapter 7 Comparison among automated methods 153
7.1 Chapter Introduction 153
7.2 The In-Parameter Order (IPO) method 153
7.3 Comparison among methods 156
7.4 Approximation of the integer programming problem by linear programming 160
7.5 Comparison with the Linear Programming approximation 165
7.6 Chapter summary 166
Chapter 8 Adaptation for practical use 167
8.1 Chapter Introduction 167
8.2 The number of values is not a prime power 167
8.3 Number of parameters "just slightly too large" 170
8.4 Dependent parameters 170
8.5 Differing numbers of values 172
8.6 Handling configurations that must be present 174
8.7 Inclusion of additional parameters 176
8.8 Inclusion of additional values 177
8.9 Chapter summary 178
Chapter 9 Conclusions and suggestions for further research 180
9.1 Chapter Introduction 180
9.2 Conclusions 180
9.3 Suggestions for further work 183
9.3.1 Differing numbers of parameter values 183
9.3.2 Numbers of parameters "just slightly too large" 185

9.3.3 Issues with block sizes 186
9.3.4 Additional initial stage building blocks 188
9.3.5 Generalization of construction algorithms to *d*-way interactions 189
9.3.6 The case with 2 values 192
9.4 Summary of future work 192
References 193

Index of Figures

Figure 2-1: A system test context 24
Figure 2-2: COTS components in an electronic commerce application 26
Figure 2-3: Class substitution with polymorphism and inheritance 28
Figure 2-4: Testing Java interfaces 30
Figure 2-5: ROOM substitutable actors 31
Figure 2-6: A Use Case Map (UCM) 32
Figure 2-7: Use case map with stub 33
Figure 2-8: Tool preference dialog 35
Figure 3-1: Construction of an interaction element 48
Figure 3-2: Interaction elements contained within a test configuration 51
Figure 3-3: Coverage of Interaction Elements 55
Figure 3-4: Formulation of a test coverage problem as an integer program 64
Figure 4-1: A system test scenario 72
Figure 4-2: A 3×3 Latin square 76
Figure 4-3: Two 3×3 orthogonal Latin squares 78
Figure 4-4: Superimposed orthogonal Latin squares 78
Figure 4-5: Three superimposed 4×4 orthogonal Latin squares 82
Figure 4-6: The orthogonal array $O(16,5,4,2)$ 89
Figure 5-1: The covering array $C(5, 4, 2, 2)$ 97
Figure 5-2: Two incomplete 6×6 orthogonal Latin squares 98
Figure 5-3: The covering array $C(37, 4, 6, 2)$ 99
Figure 6-1: Orthogonal (O), basic (B), and reduced (R) arrays 110
Figure 6-2: Examples of Z and N arrays 111
Figure 6-3: First step for constructing $C(15, 12, 3, 2)$ 112
Figure 6-4: Second step for constructing $C(15, 12, 3, 2)$ 113
Figure 6-5: The completed covering array $C(15, 12, 3, 2)$ 114
Figure 6-6: $C(15, 12, 3, 2)$ in block composition notation 114
Figure 6-7: $C(17, 16, 3, 2)$ in block composition notation 115
Figure 6-8: $C(21, 36, 3, 2)$ in block composition notation 115
Figure 6-9: First step of constructing $C(15, 13, 3, 2)$ 116

Figure 6-10: The completed array $C(15, 13, 3, 2)$ 117
Figure 6-11: $C(15, 13, 3, 2)$ in block composition notation 118
Figure 6-12: $C(21, 40, 3, 2)$ in block composition notation 118
Figure 6-13: Results from the method 148
Figure 7-1: Results from the linear program approximation 164
Figure 8-1: Elimination of redundant configuration when n is increased from 6 to 7 169
Figure 8-2: Dependent parameters 171
Figure 8-3: Construction based on n_1, with added configurations for n_0 173
Figure 9-1: $C(12,48,2,2)$ constructed in four stages 186
Figure 9-2: $C(12, 48, 2, 2)$ constructed in five stages 187
Figure 9-3: $C(25, 52, 3, 2)$ constructed using Algorithm 6-2 187
Figure 9-4: $C(23, 52, 3, 2)$ constructed using algorithm extensions 188

Index of Tables

Table 3-1: Set of selected test configurations 59

Table 3-2: Results of running lp_solve on test configuration problems 66

Table 4-1: The orthogonal array *O*(9, 4, 3, 2) 72

Table 4-2: Enumeration of parameter values 73

Table 4-3: Test configurations for the system test scenario 73

Table 7-1: Comparison of AETG, Pair-Test / IPO, and TConfig 157

Table 7-2: Comparison of Pair-Test and TConfig for time and parameter growth 159

Table 7-3: Comparison among TConfig, IPO, and the linear programming approximation 165

Table 8-1: Constructing an independent parameter 171

Table 8-2: Dependent parameter correspondence 172

Table 9-1: The orthogonal array *O*(8,7,2,2) 184

Table 9-2: *O*(8,7,2,2) modified for one parameter with four values 185

Table 9-3: Small covering arrays found by simulated annealing 189

Index of Definitions

Definition 3-1 Number of system parameters: k 43

Definition 3-2 Number of values for each parameter: n_i 43

Definition 3-3 Number of values scalar function $n = f(n_0, ..., n_{k-1})$ 44

Definition 3-4 Indefinite value v_i assigned to a parameter i 44

Definition 3-5 Definite value assigned to a parameter 44

Definition 3-6 Selection of a test configuration τ 45

Definition 3-7 Set of all possible test configurations T 45

Definition 3-8 Selected test configurations S 45

Definition 3-9 Degree of interaction coverage d 48

Definition 3-10 Index subset σ_d 48

Definition 3-11 Set of all possible index subsets Φ_d 48

Definition 3-12 Interaction elements $\chi(\sigma_d, \tau)$ 49

Definition 3-13 Set of all interaction elements X_d 49

Definition 3-14 Representation of a test configuration τ' as a set of d-interaction elements 51

Definition 3-15 Set of all possible test configurations T', expressed as interaction elements 52

Definition 3-16 Set of selected test configurations S', expressed as interaction elements 53

Definition 3-17 The interaction test coverage problem of degree d 53

Definition 3-18 Set of interaction elements covered by a set of test configurations 56

Definition 3-19 Measurement of Interaction Test Coverage of specified degree 56

Definition 3-20 The interaction test coverage problem of degree d (as a decision problem) 60

Definition 3-21 The minimum set cover problem 60

Definition 4-1 Orthogonal array $O(\rho, k, n, d)$ 71

Definition 4-2 Latin square 75

Definition 4-3 Orthogonal Latin squares 76

Definition 4-4 Galois (finite) field 78

Definition 5-1 Covering array 95

Definition 6-1 Basic array, $B(n^2 - 1, n + 1, n, t)$ 108

Definition 6-2 Reduced array, $R(n^2 - n, n, n, t)$ 109
Definition 6-3 Array of zeros, $Z(\rho, t)$ 110
Definition 6-4 1 to $n - 1$ array $N(n^2 - n, n, t)$ 110
Definition 6-5: Number of stages, s 119
Definition 6-6 Number of columns to use in array building blocks, g_r 119
Definition 6-7 Parameter capacity at stage r, h_r 119
Definition 6-8 Column duplication values, t_r 121
Definition 6-9 Number of asymmetric columns, $e_r(s)$ 122
Definition 6-10 Duplication factor for asymmetric columns 123

Index of Algorithms

Algorithm 4-1 Extension of a field 79

Algorithm 4-2 Generate a set of orthogonal Latin squares 79

Algorithm 4-3 Convert orthogonal Latin squares from Galois field to integer elements 80

Algorithm 4-4 Generate orthogonal array $O(n^2, n+1, n, 2)$ 83

Algorithm 5-1: AETG's algorithm to add a new configuration 103

Algorithm 6-1 Determine parameter capacity 120

Algorithm 6-2 Determine usage of basic versus reduced arrays 120

Algorithm 6-3 Construct $C(\rho, k, n, 2)$ 125

Algorithm 7-1 The In-Parameter Order (IPO) Algorithm 155

Index of Lemmas and Theorems

Lemma 4-1 Contents of row 0 86

Lemma 4-2 Contents of first n rows, except for column 0 87

Lemma 4-3 Contents of column 0 87

Lemma 4-4 Contents of each column, except column 0 87

Lemma 6-1 Coverage of B array 108

Lemma 6-2 Coverage of R array 109

Lemma 6-3 Value of t_r 121

Lemma 6-4 Value of $e_r(s)$ 122

Lemma 6-5 Value of λ_{qr} 124

Lemma 6-6 Columns in S after concatenations of A 126

Lemma 6-7 Columns in S after concatenating R and N arrays 127

Lemma 6-8 Concatenations involving array C, and number of columns in C 129

Lemma 6-9 Number of stages required 131

Lemma 6-10 The array C has a sufficient number of columns 131

Lemma 6-11 Results when modulus increases 134

Lemma 8-1 First row of covering array 175

Theorem 4-1 Algorithm 4-2 generates a set of $n - 1$ orthogonal Latin squares 79

Theorem 4-2 Algorithm 4-4 generates an orthogonal array 83

Theorem 6-1 The array C is a covering array of strength 2 134

Theorem 6-2 Complexity of constructing $C(\rho, k, n, 2)$ 149

Glossary of Variables

B	basic array
C	covering array
c_d	interaction coverage of degree d
d	degree of interaction coverage
$e_r(s)$	number of asymmetric columns at stage r, based on s total stages
$f(\,)$	scalar function for the number of parameter values
GF	Galois field
g_r	number of columns to use in array building blocks at stage r
h	parameter capacity
k	number of parameters
$\overline{L}$	set of orthogonal Latin squares with entries from a Galois field
L	set of orthogonal Latin squares with integer entries
L^l	Latin square l
m	power of prime in Galois field order
n	scalar representing "number of values" for array constructions
N	array containing zeros through $n-1$'s
n_i	number of values for parameter i
n_0	maximum number of parameters for any value
n_1	second largest number of parameters for any value
O	orthogonal array
p	prime factor used to generate Galois field
R	reduced array
s	number of stages required to construct covering array

S .... set of selected test configurations

S' .... set of selected test configurations expressed as interaction elements

T .... set of all possible test configurations

T' .... set of all test configurations expressed as interaction elements

t_r .... number of times to duplicate columns at stage r

v_i .... the value selected for parameter i

X_d .... set of all possible interaction elements

x_i .... ith element of Galois field

Z .... array containing all zeros

α .... element used to construct Galois field

λ .... factor for repetition of N arrays

Φ_d .... set of all index subsets of cardinality d

$\chi(\sigma_d,\tau)$ .... interaction element, based on test configuration τ and index set σ

ρ .... number of rows (test configurations) in an orthogonal or covering array

σ_d .... index subset of cardinality d

τ .... a single test configuration

τ' .... a single test configuration expressed as a set of interaction elements

Chapter 1 Introduction

This thesis addresses the problem of testing interactions among components of a software system: the "interaction test coverage" problem. In particular, we address the problem of testing software systems in situations where the number of test configurations is so large that testing of all configurations is not possible within any realistic budget of time or money.

There are two complementary sides to the overall problem of testing a software system. Both problems must be addressed adequately. The first problem is to create a test suite, given a description of the expected behaviour of a system configuration. Much of the literature in software testing has dealt with this problem. For introductory texts on software testing, see [My76], [Bei90], [Mar95] or [Bi00]. Methods for automatically generating tests have focused on how one might generate tests from a formal specification of behaviour (for example, see [Ur93], [Pr99], [Sa00], or [Pr01]).

The second problem is to deal with a large number of independent test configurations. This problem typically arises at the system test level, although it can appear at even the unit test level. Examples include setting up network elements in a distributed system, and selection among a set of software configuration options. It is when independent components are integrated or assembled that interactions typically become a problem.

We address the second problem in this thesis: the situation when there are various system parameters, each of which can take on a value from a discrete set. Furthermore, the parameters are independent, in the sense that setting a parameter to a particular value has no influence over what values other parameters may take. In addition, setting a parameter to a particular value has no influence over the existence of other parameters (for example, choosing a type of payment method is only relevant when payment is necessitated). If there are dependencies among parameters, they can be resolved using the approach described in section 8.4.

To motivate the study, we look at a range of situations where the interaction test coverage problem arises in practice. Previous work in this area is also discussed.

Next, we formally define the interaction test coverage problem, and give it a theoretical framework. In particular, the problem is shown to be related to the minimum set cover problem and the {0, 1} integer programming problem, although there are some distinguishing features with respect to both problems. We also define a metric for measuring interaction test coverage, so that the quality of a set of system test configurations can be evaluated.

Next, we introduce techniques from statistical experimental design. These techniques are used for interaction testing in other fields such as chemistry and medicine to detect desired or undesired interactions among processes or treatments. The approach is also becoming noticed for quality control in engineering.

We look at how to adapt these techniques to test software. Some attributes of software testing provide for optimization of experimental designs. We discuss why this is possible, and why the use of a "covering array" can result in fewer test configurations to achieve a specified level of interaction coverage.

We then discuss how to construct covering arrays. An algorithm is proposed that is deterministic, and does not have some of the limitations that are inherent in standard experimental design techniques. (The limitations will be described in section 5.2).

The algorithm is then compared with other methods, to show that it is much faster, can handle larger problems, and in most cases, produces fewer configurations.

There are several issues in applying the theory to real, practical, situations, and we then discuss methods for addressing these issues. The result is a method that can be applied in general cases.

Finally, we identify areas that would benefit from additional research to further improve the methods proposed herein.

1.1 Contributions of this thesis

This thesis investigates several questions about system interaction test coverage: What is it? How can it be measured? How can coverage be achieved? When does the situation arise in practice?

The major contributions of this thesis are as follows:

1. A formal definition of the interaction test coverage problem is presented. [Section 3.2.3]
2. A metric for measurement of interaction test coverage is defined. [Section 3.3.1]
3. The interaction test coverage problem is related to some well-known combinatorial optimization problems, and the NP-completeness of the problem is discussed. [Section 3.4]
4. The relationship is developed between software system interaction testing and statistical experimental design, in terms of the additional properties of software system testing that allow us to use a smaller experimental design. [Section 5.2]
5. A new, fast, deterministic, algorithm is developed for achieving pair-wise interaction coverage. [Section 6.2]
6. The new algorithm is implemented, and compared with other methods. [Sections 7.3, 7.5]

1.2 Notation conventions

In this paper, we shall consistently use notation where:

- A set of n elements will be numbered from 0 to $n - 1$.
- All matrix row and column indices will be numbered from 0 to $n - 1$.

The reason for this is that much of the mathematics in this paper will be based on modular arithmetic, and it is more convenient to adopt this convention so that values will correspond with the range of values resulting from modular arithmetic.

We shall also use the following assumptions on summation and product notation. If a summation, denoted $\sum_{i=a}^{b} x_i$, results in having no valid terms because $a > b$, then the value of the summation is deemed to be zero. Similarly, if a product, denoted $\prod_{i=a}^{b} x_i$, results in having no valid terms because $a > b$, the value of the product is deemed to be one.

Chapter 2 Software Component Interaction Testing in Practice

2.1 Chapter Introduction

A common source of software and system faults is unexpected interactions among system components. The risk increases when for each element in a system, there are a number of interchangeable network components. A manufacturer of these system components would want to test as many of the potential system configurations as possible, to reduce the risk of interaction problems. However, the number of potential system configurations grows exponentially (see section 3.2.1).

The construction of a test suite for a single system configuration is a problem for which there is an extensive literature (introductory texts are [My76], [Bei90], [Mar95] or [Bi00]), and therefore is not discussed in this work.

We do note that the adaptation of a test suite to run in various configurations is an additional problem, as is the reconfiguration of the system itself. However, these problems usually depend strongly on the particular test environment. Nevertheless, these may require non-trivial amounts of effort, and are incentives to keep the number of test configurations as small as possible.

System testers face constraints on time and money, and it is quite likely that they cannot test all possible configurations within any realistic budget. How, then can we manage the risk of interaction faults in a realistic test plan?

In this chapter, we present eight examples of contexts where testing for unwanted interactions among parameters can be performed. This is by no means an exhaustive list. The examples are organized into the following categories:

- Substitutable components
    - Network component interaction testing

- Commercial off-the-shelf (COTS) components
    - Real-time Object-Oriented Methodology (ROOM) substitutable components
    - Stubs in Use Case Maps (UCMs)
- Object-oriented cluster testing
    - Testing in the presence of polymorphism and inheritance
    - Testing of Java interfaces
- Test data selection
    - Equivalence class partition testing
- System configuration parameters
    - User configurable tool parameters

The examples motivate discussion of how to generate the test configurations to detect unwanted interactions, by showing that the results can be used in many situations. This shows that the generation of test configurations for interaction testing is a general problem, where our solutions can be applied widely.

2.2 Substitutable Components

In this section, we look at situations where each parameter is a component for which there are a number of variants that can be substituted.

2.2.1 Network Component Interaction Testing

In a telecommunications network, there are frequently a large number of allowable configurations. In particular, a network may consist of various components, and each component may have several different types that can substitute freely for one another. For example, in a telephone network, the calling telephone might be an ordinary residential phone, a cellular phone, or a pay phone. A manufacturer of telecommunications

equipment may also produce telephone switches intended for various markets. While not strictly a network component, another factor is the type of call to place. Figure 2-1 illustrates a possible context.

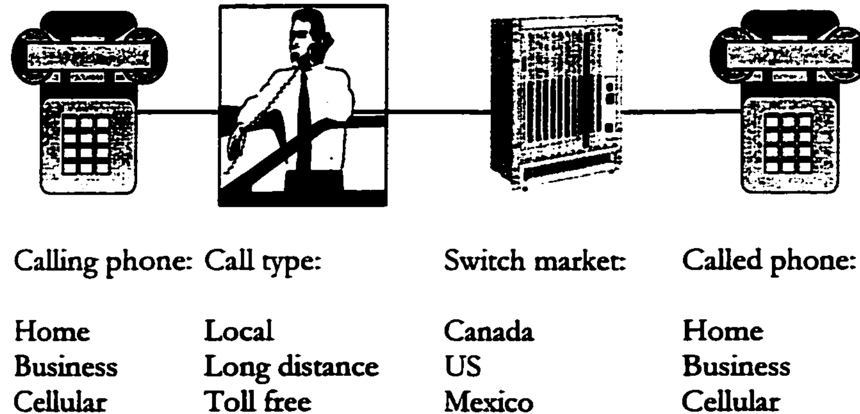


Figure 2-1: A system test context

We are assuming that a suitable test suite already exists for testing each individual configuration, based on the expected behaviour of the system. For example, one could have a test suite containing approximately twenty test cases covering the various aspects of a simple telephone call [Pr99]. Variations are: who hangs up first (caller or called), calling a busy telephone, abandoning a call at various points, etc. Some of these tests may vary slightly among configurations (for example, the directory number of the called telephone). At a coarse level, the expected behaviour of a simple telephone call is similar among configurations.

There may be additional tests required in some configurations. Business telephones often have a more complex display screen, and these additional features must be tested as well. Testing cellular hand-offs is peculiar to a wireless system. The result is that while some tests are invariant over various configurations at the coarse behaviour level, additional specific tests may be required for particular configurations.

During a phase of testing late in the software development process such as system testing, we assume that a test suite, hopefully as complete as possible, is available for any configuration. In practice, priorities are typically assigned to configurations that are recommended to customers, and therefore, the test suites are geared towards those

configurations. In the situation shown in Figure 2-1, adapting the test suite may be as simple as changing the phone numbers used to make the calls. In other situations, extensive test suite modifications may be required to change configurations.

A system tester for networks of realistic size faces a large number of possible system configurations. The trade off that the system tester must deal with is the thoroughness of test coverage, versus the limited resources of time and budget that are available.

It may be the case that customers can configure the network for their purposes. That is, it may not be possible to predict in advance which configurations customers will select, and concentrate on testing those configurations. Therefore, we need a strategy to achieve the best coverage possible within a limited test budget. We will develop such a strategy in section 3.2.3.

2.2.2 Commercial off-the-shelf (COTS) components

Software providers are increasingly building their products by including commercially available components, in order to decrease their time to market [Er99].

There are two types of usage of COTS components. The first is the re-use of infrastructure that is already in place. The second type is to incorporate the components into an integrated system.

An electronic commerce application is an example of the first type of COTS usage. Much of the infrastructure is already in place: the internet for communication; browsers for the clients' user interface; corporate data bases for merchandise catalogues and recording of transactions; electronic payment servers for various credit cards; and so on. Indeed, it would be desirable to work with the browsers of various manufacturers, and even with various versions of those browsers. Figure 2-2 shows an example.

An application that processes electronic commerce transactions is a particularly appropriate example because the application would have generally similar functionality over a range of system architecture environments. Therefore, this type of testing is especially important for such applications.

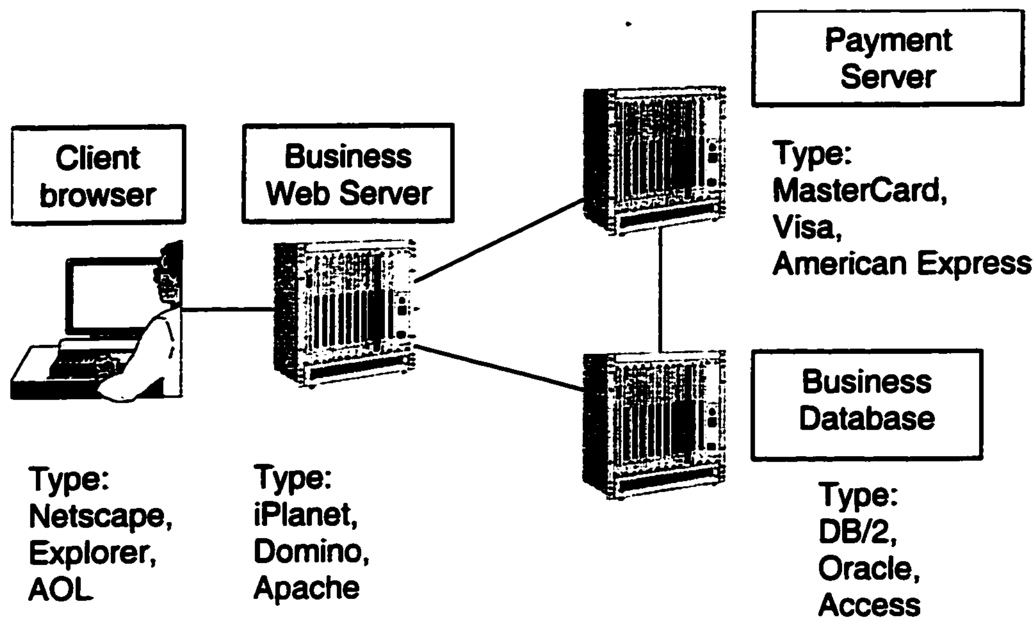


Figure 2-2: COTS components in an electronic commerce application

Another example of the first type of COTS application is the Telelogic Tau tool set [Te]. This tool provides a real-time software development environment. Tau is based on Message Sequence Charts (MSCs) [ITU00a] for requirements modelling, Specification and Description Language (SDL) [ITU00b] for specification and design modelling, and Tree and Tabular Combined Notation (TTCN) [ISO92] for describing abstract test cases. To provide executable simulations of SDL models and TTCN test cases, the tool requires the use of a C or C++ compiler. Inside the Tau tool, the user requests that a simulation is performed, and after the simulation is constructed, it is executed using the simulator user interface. While the user has continued to work inside the Tau interface, an external compiler has been invoked. The compiler is not provided, at present, with Tau. Instead, there is an extensive wrapper around the compiler environment to connect Tau with the external compiler. (A "wrapper" is a software pattern which is typically used to enclose a software component, such that a standard interface is presented to the enclosing software. If a new version of the component appears, the modifications required to the program should be restricted to within the wrapper. This is an instance of the "adapter" pattern as described in [Ga95].)

In turn, the compiler relies on a set of C or C++ libraries, some of which are provided by Tau, and some of which are on the system. Tau contains a number of individual tools that require the compiler: the SDL simulator, the SDL validator, the SDL-TTCN link, and the TTCN simulator to name a few. Each of these has a wrapper to the compilation environment, and library functions. With the number of tools available, the number of wrappers required to interface with various compilers, and the various sets of compiler libraries, the number of configurations has become very large.

In the second approach, the COTS components are used as building blocks to produce a new product. Usually, a software "wrapper" is put around each component, so that the desired functionality filters through to the complete product through a consistent interface. Undesired or extra functionality of the component is blocked from interfering with the complete product. The product is then constructed by providing "glue" code that connects the wrapped components in such a way to provide the functionality of the new product.

One of the issues that arise using this approach is, since the components are provided by other vendors, the release schedules of new versions of the components are independent of the overall product. It is a product management challenge to determine which versions of each component are acceptable for use in the complete system. Therefore, interaction testing is required to determine the set of workable configurations of the system.

To perform this interaction testing, we again have the situation where there are a number of *parameters* (the software components). Each of these parameters has a number of *values* (the component versions) that can be set independently. (These values will be defined more precisely in section 3.2).

2.3 Object-oriented and Use-case Based Testing

With the advent of object-oriented design and programming, the structure of a software system has become a set of encapsulated objects that communicate by invoking methods in each other. This assembly of components may result in interaction problems.

One type of component assembly occurs with the availability of a class library or framework, which may be developed in-house or purchased from a third party. New systems are constructed by assembling component classes in ways that the designers of the components may not have envisaged originally.

2.3.1 Testing in the presence of polymorphism and inheritance

A possibility for the substitution of software components comes in the presence of an inheritance hierarchy. Instances of different classes within the hierarchy can often substitute freely for each other. This is especially true when class interfaces are similar and polymorphic invocation of methods is used [Mc94]. Figure 2-3 illustrates this situation.

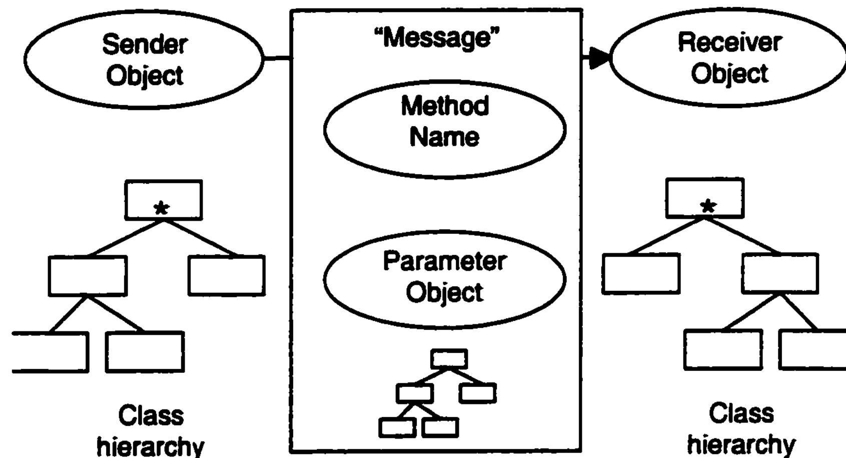


Figure 2-3: Class substitution with polymorphism and inheritance

In this situation, the sender object can be an instance of any class below a particular node in the class hierarchy (for example, the classes marked with a * in Figure 2-3). This could easily arise as specialization of a generic "phone" class, for example (compare with Figure 2-1). Similarly, the receiver object could be an instance of any class below a particular node in the class hierarchy. If the message to be sent between sender and receiver includes parameters, these objects may also be instances of a range of classes in the overall hierarchy. Again, we have the situation where we have a number of parameters (the various objects), and a set of possible values (the specific class) for each parameter.

McDaniel and McGregor [Mc94] observed that orthogonal arrays might be a useful strategy to attack this problem, and that pair-wise coverage would be a potential test coverage criterion. However, this approach is not mentioned in a recent text on testing object-oriented systems [Bi00].

Other factors that should be considered are the "states" of the various objects involved. This could be an explicit state in a state-based design model, but it can also refer to the current values (or equivalence classes thereof) of the encapsulated data. Another factor is the selection of method invoked in the receiver object. These factors come with a number of dependencies. Legal messages that are accepted and the message parameters most likely depend on the state of the receiving object, and may depend on the state of the sender as well. All of these factors result in a complex testing scenario.

We should address the question of whether complex inheritance hierarchies are used in practice. The advent of object-oriented class libraries and application frameworks can lead to deep inheritance hierarchies. This is particularly true for frameworks for the development of graphical user interfaces. For example, one commercially available C++ development environment [Sy] for the Macintosh computer is structured so that an application program should be derived from a class that is five levels deep in the class hierarchy. The result is an application that retains the standard "look and feel" of a Macintosh application. The main program for a document file is derived from a class eight levels deep. The classes for the graphical interface are where multiple levels of hierarchy become relevant. All windows are to be derived from a class "CWindow", and dialogs are to be derived from a class "CDialog" that is itself is two layers beneath CWindow in the class hierarchy. The user's manual for this environment specifically indicates situations where the user is expected to override class methods of a parent class. Therefore, testing for the effects of polymorphism and inheritance is particularly important in this situation.

2.3.2 Testing Java interfaces

In the Java language, the concept of implementation classes has been incorporated into the Java language, but using the mechanism of defining a Java *interface* as well as

inheritance. An interface allows a user to specify a set of operations that a class instance must perform, without actually providing classes that satisfy the interface. However, any classes that are declared to belong to the interface must provide an implementation for each method call declared in the interface.

For example in the "java.io" standard library package [Su], a "DataInput" interface is defined. Classes related to the data input operation should conform to this interface. Two classes that belong to the "DataInput" interface are "DataInputStream" and "RandomAccessFile". Each of these classes must provide an implementation of the method calls exactly as specified in the interface.

A situation that might arise is that an instance of a class conforming to the "DataInput" interface could call a method that is in the "DataOutput" interface, and pass parameters that are instances of a class that implements the "Serializable" interface (has a conversion to binary format capability). This is shown in Figure 2-4:

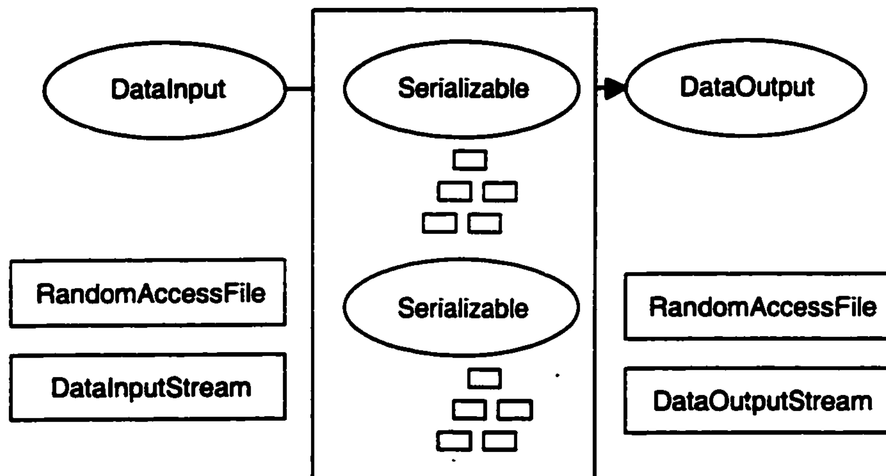


Figure 2-4: Testing Java interfaces

To ensure proper testing of this scenario, a strategy to deal with the large number of configurations is needed. The parameters are the interfaces described in the scenario, and the values for these parameters are the classes that implement the interfaces. Once again, a large number of test configurations may result.

2.3.3 Real-time Object-Oriented Methodology (ROOM) substitutable actors

A specific example of object-oriented components occurs in the Real-time Object-Oriented Methodology (ROOM) [Se94], used in the Rational Rose Real-Time tool set. In ROOM, there is the notion of a *substitutable actor.* An *actor* in ROOM corresponds to a software component that has encapsulated behaviour defined using hierarchical extended finite state machines. A ROOM actor is represented graphically (see Figure 2-5) by a rectangle. An actor's *interface* is defined by a set of *ports*, through which messages are sent or received. Ports have *protocol classes* associated with them. A protocol class defines the set of messages that can be sent or received through the interface port. A port is illustrated as a small filled square on the boundary of an actor. Messages are communicated between actors along *bindings*, represented by lines connecting ports.

A substitutable actor provides the capability to substitute any actor that corresponds with the template defined by its interface ports into a set of components. For each substitutable actor, one can insert any of the set of available actors that fit the interface. The result is an exponential set of possible configurations. In ROOM, a substitutable actor is indicated by a cross in the upper left corner. Figure 2-5 illustrates this situation.

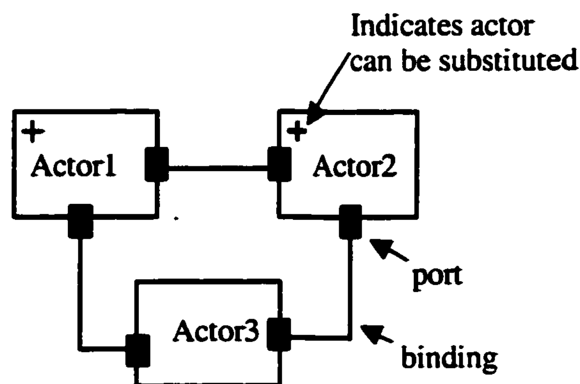


Figure 2-5: ROOM substitutable actors

In Figure 2-5, there is a system configuration with three actors, and various communication paths among them. The two actors marked as substitutable, Actor1 and Actor2, can have different implementation actors plugged in to that location in the configuration. The substitutable actors represent system configuration parameters, and the actual actors that can be used represent discrete values for those parameters. For

example, if there are three actors that can be inserted for Actor1 in Figure 2-5, and two actors that can be inserted for Actor2, then there are six possible configurations.

2.3.4 Stubs in Use Case Maps (UCMs)

Use Case Maps [Buh96] are a high level method for capturing requirements. A UCM aims to capture the flow of responsibilities among components, as one would trace a finger through a diagram. Figure 2-6 shows a UCM with three components and five responsibilities, "a" through "e," which must be handled among the components. The thick line starting with the filled circle, and continuing through to the bar at the end, shows the order of responsibilities, and which component handles each responsibility.

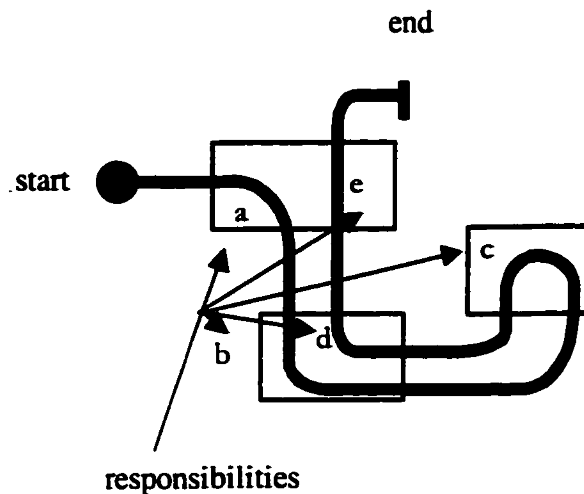
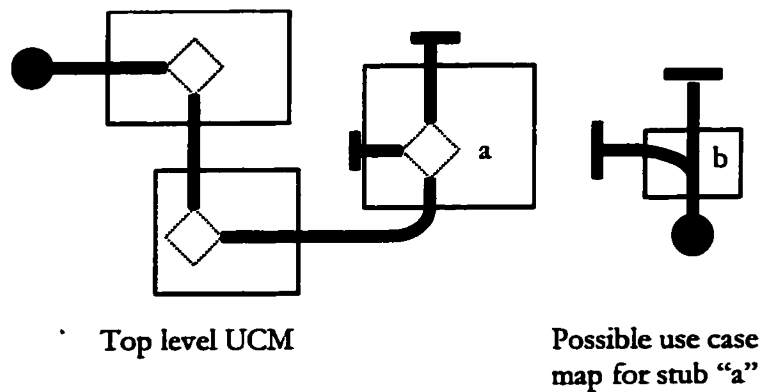


Figure 2-6: A Use Case Map (UCM)

An important feature of use case maps is "stubs". A stub indicates a point where a sub-UCM can be inserted. The sub-UCM can be any UCM compatible with the large-scale UCM, and is used as an equivalent to a procedure call or to expand detail. Again, we have the situation where a number of components can be assembled, where there are multiple versions of each component available. Figure 2-7 shows an example of a top-level UCM, and a stub that could be substituted within it.



Top level UCM

Possible use case map for stub "a"

Figure 2-7: Use case map with stub

UCMs with stubs have been used in a project [Buh98] to specify a communications system using dynamic agents. Because of the large number of agent-feature interactions, coverage of pair-wise combinations of agents and features was used as the test coverage goal.

The stub insertion locations represent system configuration parameters, and the set of stubs that can be inserted at those locations represent the values for those parameters.

2.4 Test data selection

In this section, we will look at situations where test input data consists of discrete values, or can be formed into a set of equivalence classes.

2.4.1 Equivalence class partition testing

One approach to creating tests is to determine equivalence classes for input and output values of software [My79] [Bei90]. Based on the relations contained in a program or specification branch, two equivalence classes of values for each variable are induced by an "if" statement: those resulting in a "true" result, and those resulting in "false."

For example, the C program statement `if (x > 3)` induces two equivalence classes for the value of x: $(-\infty,3]$ and $(3, \infty)$. If this was the only statement involving the variable `x` that affects control flow, then selecting one value from each of these two equivalence classes would cover all control flow dependencies for this variable.

If the type of branch is a case or switch, there can be many equivalence classes. Definition-use associations can be used to connect the branch relation to a set of input values. After considering all program or specification branches, we can derive a set of equivalence classes for each of the input variables, where using any of the values in an equivalence class as input will result in following a similar execution path.

The strategy is to make sure that each equivalence class for each variable is covered at least once by a test case. Boundary, "on by one," and "off by one" values for the equivalence classes – the values that are on the class boundary, just inside the class, and just outside the class – should also be examined (see [My79] or [Bei90]). For example, if we have the equivalence classes $(-\infty,3]$ and $(3, \infty)$ from the condition $x > 3$, and x is an integer, then it would be prudent to have test cases where $x = 2, 3, 4$. For robustness testing, the boundaries at MAXINT, (the maximum value for the variable type) and the negative equivalent should also be considered.

However, this approach does not directly address the possible interaction of equivalence classes among various input parameters. This is particularly important to test compound relations in branching statements. Given a set of variables, for each variable, we have a set of equivalence classes. This corresponds to a set of parameters, each of which has a discrete set of independent "values". The process of finding equivalence classes for each input value converts a large input domain to a smaller set of discrete items. Therefore, we can use a similar approach to handle it: attempting to detect unwanted interactions among equivalence classes for different parameters. For example, if the first parameter to a function has five equivalence classes for its data values, and the second parameter has three equivalence classes, an objective is to try each pair-wise combination of the equivalence classes for the two parameters.

Pair-wise coverage of equivalence classes seems particularly appropriate. Consider the construction of conditional branches in a program. Conditions involving comparison of a single variable with a constant are commonly used (for example, iteration). Comparisons between two variables are also reasonably common, and the latter is the situation where covering pair-wise interactions of equivalence classes is valuable.

How often does a comparison using three or more variables occur in a single condition? In the author's experience, these are used only rarely. Of course, if a designer has used such a complex condition, it may be prone to errors. Testing of three way (or more) interactions of equivalence classes is advisable when a software designer has knowingly used such a condition. Scanning code for complex conditions could be useful to determine the degree of risk associated with covering only pair-wise interactions of equivalence classes.

2.4.2 User configurable tool parameters

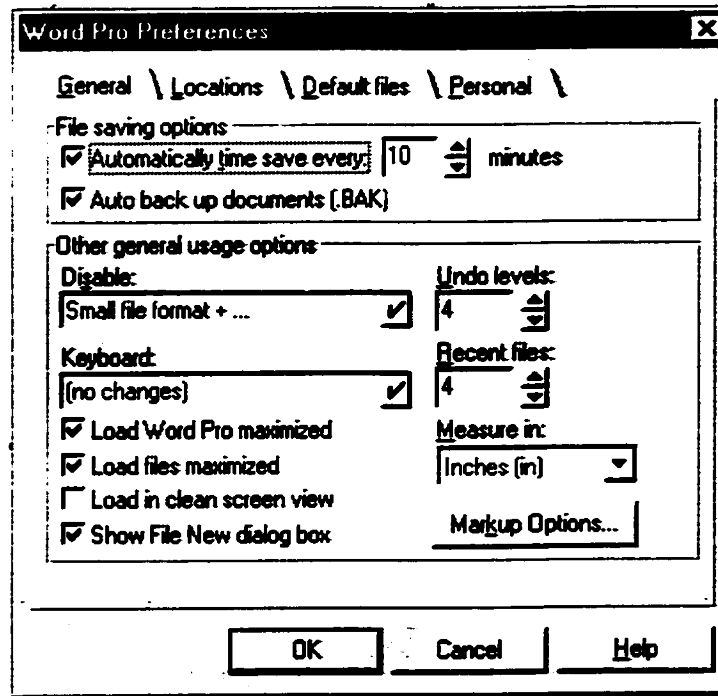


Figure 2-8: Tool preference dialog

Another situation where a large number of configurations may be present is in configurable user applications, especially those with graphical user interfaces [Wh96]. For example, if you select the menu "File - User Setup - Word Pro preferences" in Lotus Word Pro 97 you will see the dialog shown in Figure 2-8.

In this dialogue alone, there are eleven independent parameters, and one dependent parameter (the time for automatically saving the file). The check boxes each represent a parameter with two values. For parameters with more than two possible values, drop-

down lists are used. For example, the "Measure In:" list has the four options of inches, centimetres, picas, or points. This dialogue also has four tabs, and the above description is only for the "General" tab.

There is the potential for interactions among all of these parameters. For example, the user can independently set the "disable small file format" and the "load files maximized" parameter values. Because they both deal with the reading of files, the tool has to work in all possible combinations of these parameter values.

The nature of a graphical interface, where there are a number of menus, each with a number of entries, presents a large number of possible selections, and a system configuration test selection strategy is needed for thorough testing.

2.5 A reasonable approach: coverage of pair-wise interactions

A possible trade off between test coverage and the number of tests is to determine a set of test configurations that tests each pair-wise combination of parameter values. (The approach can be extended to three-way interactions, and so on.) That is, for any two parameters in the system, each possible combination of the possible values of the two parameters occurs in at least one test configuration.

This goal provides a well-defined level of test coverage, with a reduced number of system configurations. Pair-wise parameter coverage can detect any problem with pair-wise compatibility of parameters, assuming the test suite for any configuration has thorough coverage. It also guarantees that all interfaces have been exercised. However, pair-wise parameter coverage cannot guarantee detection of problems involving a specific combination of three or more parameter values.

In the next section, we look at previous applications of this approach, most of which investigate pair-wise coverage of parameters.

2.6 Previous work on pair-wise (or *d*-way) interactions

[Note: Some of the papers in this section are also discussed in section 5.4.2, in cases where the paper contains discussions of tools as well as applications. The commentary in this section is limited to applications of a pair-wise (or *d*-way) coverage criterion.]

The notion of pair-wise coverage of parameter values in software testing appears first in a 1985 paper by Mandl [Man85] for testing an Ada compiler. In particular, the application that is used as an example is testing the types of operands as part of equations. The parameters are arithmetic binary operators (for example, +, or –), and the type of its two operands (such as integers, long integers, or floats). To generate pair-wise coverage of the parameter values, the method of orthogonal Latin squares is introduced. Mandl also presents some of the restrictions imposed by the theory on which the method is based. The conclusions of this paper are mostly related to the "comfort" of having a systematic method of generating test suites, as opposed to a random selection.

Brownlie, et al. (1992) [Br92] report on a case study of testing the AT&T PMX/StarMAIL system using orthogonal arrays. In this case, the method is described as "Robust Testing." A tool called OATS (Orthogonal Array Test System) was used to generate the orthogonal arrays. The paper focuses on how the large number of system configurations, and short testing interval led to the first application of orthogonal arrays to achieve pair-wise coverage of configurations at the system test level. The paper reports that the major benefit for the group doing the testing is that the original test plan for the system, developed using "conventional" methods, required more resources than were available. In particular, the number of tests in the original plan was too large for the time available – and the original test plan was never executed because of this. However, an analysis after execution of the "robust testing" suite showed that 12% of the faults found using this method would have been missed by the original test plan. Furthermore, because the original test plan could have only been partially executed, the authors estimate that a further 10% of the faults found would have been in test cases that would have to be dropped for time constraints. Aside from finding 22% fewer faults, it was

estimated that the original test plan would have used double the number of staff-weeks for test suite development and execution time.

A number of experience reports are available on the use of Telcordia's (formerly Bellcore) AETG (Automatic Efficient Test Generator) system, developed by Cohen, Dalal, Parelius, and Patton. These reports show how the method can be applied, and give some encouraging results in support of the notion of pair-wise coverage.

AETG itself is partially described in [Coh94a], [Coh94b], [Coh96], and [Coh97]. The discussion of the AETG tool and its methodology will be deferred to section 5.4.2. The results on the effectiveness of the testing using AETG is presented below.

The first report [Coh94a] (1994) introduces the idea of a combinatorial approach to test generation, based on statistical experimental design techniques. The application to be tested is the user interface to a Telcordia database; in particular, the user input screens. The use of the system is then described in that the user has to enter the set of fields (parameter values) and relations (dependencies among parameter values). It is noted that AETG can also generate test configurations for *d*-way interactions, and not just pair-wise interactions.

The second AETG report (1994) [Coh94b] shows the applicability of AETG to testing of the Advanced Intelligent Network (AIN) voice response unit. Descriptions of applicability to telecommunications feature conformance, interoperability, and protocol conformance testing are provided.

The third AETG report (1996) [Coh96] contains a study of applying pair-wise coverage of input data equivalence classes to the Unix 'sort' command. The results are:

- Block coverage: 93.5%
- Decision coverage: 83%
- Computational uses: 76%
- Predicate uses: 73.5%

In the fourth AETG report (1997) [Coh97], AETG was again used to construct a test suite that had pair-wise coverage of input data equivalence classes. Test suites were constructed for two releases containing 9 and 13 modules of code, with approximately one to two thousand lines of C code per module. For both releases, a conventional test plan had already been executed. In the first release, an average of 6 additional defects per module were discovered, and in the second release, an average of 4 additional defects were discovered. In another experiment, 300 test cases of random input data were developed for a system. Measurements of code coverage were taken for the random test suite, and then the 200 tests developed by AETG. The tests developed by AETG had 25-35% better control flow coverage and 10-15% better data flow coverage than the random test suite. Furthermore, 436 tests with all valid input combinations were run, and the test suite produced by AETG had exactly the same code coverage.

An AETG experience report is by Burroughs, et al. (1994) [Bur94]. The method is used to set up system configurations for primary rate ISDN; in particular, call rejection and channel negotiation. In the first application to test call rejection, there were five parameters, with 7, 6, 3, 2, and 2 values each. In the second case of testing channel negotiation, there were four parameters with 3, 6, 3, and 2 values each. The benefits described in this paper focus on the benefits of having a systematic approach to eliminate large numbers of potential configurations.

Another AETG experience report is provided by Dunietz, et al. (1997) [Dun97]. The paper reports on an experiment where code coverage measurements are taken for d-way parameter interactions, for $d = 1$ up to k, the number of parameters. The result was that pair-wise coverage generally provided the equivalent of block coverage. To achieve the same path coverage as an equivalent number of randomly generated tests, four-way interactions had to be covered. This report does not give any data on faults found.

McDaniel and McGregor (1994) [Mc94] investigate the potential of applying pair-wise coverage to object-oriented testing, in the presence of polymorphism and inheritance. Suppose that an object sends a message to another object. The class of the receiver may be any class in the class hierarchy for which the message is valid. Furthermore, the

receiving object may have a number of internal states, for which receiving the message causes different behaviour. Then we have to select a sender of the message, and its class and object state may have similar sets of possible values. Finally, the message parameter objects are instances of a set of classes, and which have internal states. Testing all of these combinations can quickly become impractical, and the goal of pair-wise coverage of parameter values is recommended. This technical report does not go into the determination of orthogonal arrays, but uses ones available in statistical tables (such as [Col96]).

White [Wh96] has also applied the same approach to testing graphical user interfaces. The configurations generated by using orthogonal Latin squares are compared with direct enumeration of combinations, and randomly generated configurations.

Ryu, et al (2000) [Ry00] have applied the approach to testing the data part of the Transmission Control Protocol (TCP). Traditional experimental design techniques were used in this study. In the TCP protocol, there are message-specific data parameters (for example, the current flow control window size), and other parameters that affect the overall performance of TCP. In this work, several "typical" discrete values were selected for significant TCP parameters, and the pair-wise combination method was used to generate a set of test cases so that every combination of two parameter values appeared in at least one message. Using this approach found "several" system errors, one of which is described in detail in the paper. This application did not attempt to determine equivalence classes for the values of TCP parameters, however. Some of the values that were chosen were extreme values, which is a good choice for boundary testing purposes. However, some of the values were chosen to be representative of normal usage, and there was no analysis performed to investigate what sort of data equivalence classes there might be. Doing this analysis would have improved this study. However, it is interesting to see the use of pair-wise interactions as a criterion for selecting protocol data field test values.

Overall, the idea of using pair-wise (or *d*-way) interactions among parameters as a coverage strategy for system testing has found to be useful in practice in a variety of situations. The study by the creators of AETG [Coh96] has produced especially good

results with their code coverage measurement. However, as we shall see in section 5.4, there has not been as much published work as to how to determine the test configurations.

2.7 Chapter summary

In this chapter, we have looked at a number of situations where one of the issues in testing is the large number of test configurations. In particular, there are a number of parameters, each of which can be set independently to one of a discrete set of possible values for that parameter. The situation is distinguished from another testing issue, namely testing individual system configurations.

For testing individual system configurations, there is a need to determine a test suite based upon any or all of requirements, specification, design, and code structure for the implementation. Test suites can be constructed based on elements of structure or data flow.

For testing groups of system configurations, the process is to repeat a test suite (presumed to be already available) in a number of different configurations, in order to detect unwanted system interactions among system components. The system is expected to function in a generally similar (although not necessarily identical) manner across all configurations. To change configurations, the test suite would be modified only partially. For example, a different set of options may have to be set up in advance of running tests for each configuration. Network addresses or phone numbers over a network may be specific to a particular configuration. We are looking at the situation where the behaviour is consistent enough that only partial modifications of test suites are required to change configurations. The significant effort required to change configurations, for system set up and test suite modifications, is a motivator for minimizing the number of configurations.

In this chapter, we have looked at eight such situations, to show that the issue of setting system parameter configuration value occurs in a variety of contexts. In the next chapter, we will turn to providing a precise definition of the general problem.

Chapter 3 The Interaction Test Coverage Problem

3.1 Chapter introduction

In this chapter, we formally define the interaction test coverage problem, and a metric for measuring interaction test coverage. We then compare this problem with well-known combinatorial optimization problems, to highlight the specific properties of the interaction test coverage problem.

To define the problem, we introduce the concept of *interaction element*, which will serve two purposes. The first is to help provide a definition of the goal of interaction test coverage, and the second is to provide a coverage metric.

To compare the interaction test coverage problem to various combinatorial optimization problems, we first present an argument that formulating the problem using graphs does not appear to provide any insight. Instead, we show that the interaction test coverage problem is related to the *minimum set cover problem*, although there are some properties of the former that distinguish it from the latter. The interaction test coverage problem is also formulated as a constraint-based problem. We also identify properties of the interaction test coverage problem which distinguish it from the $\{0, 1\}$ integer programming problem.

Our conclusion is that the status of the interaction test coverage problem (with regard to whether it is NP-complete or not) is still open.

Portions of the material in this chapter have been published in [Wi01].

3.2 Selecting test configurations to cover parameter value interactions

In this section, we give some background to facilitate the formal definition of the interaction test coverage problem. We subdivide the problem into two parts. The first part is the general problem of selecting an ordered tuple (a configuration) from a set of discrete parameter values. The second part is to look at the problem from the point of

view of coverage of sets of interactions. We then formally define the interaction test coverage problem, and an interaction test coverage metric.

3.2.1 Selection of test configurations

We start by introducing terminology that will help to define precisely the test interaction coverage problem. The basic inputs to the problem are the number of parameters, and the number of values for each parameter.

Definition 3-1 Number of system parameters: *k*

Let k denote the number of independent system parameters. □

The property of independence means that the selection of a particular value for one parameter does not affect the existence of other parameters, or the selection of any other parameter values. A discussion of how to eliminate dependencies among parameter values will be presented in section 8.2.

Definition 3-2 Number of values for each parameter: n_i

Each parameter has a set of discrete values that the parameter can take. For i such that $0 \leq i < k$, let n_i denote the number of values for parameter i. Without loss of generality, assume that the parameters are ordered so that $n_0 \geq n_1 \geq \ldots \geq n_{k-1}$. □

It is useful to identify the largest, and second largest, number of values for any parameter, and indexing the parameters by their relative number of values allows us to easily reference these values.

Although it is not strictly required, we can assume that for any i, $n_i \geq 2$. A parameter with only one possible value is not of particular interest. We also restrict all values n_i to be positive integers; in practice, these are usually "small" integers (< 20).

It is occasionally convenient to have a single scalar value representing "the number of values," as opposed to dealing with a vector of quantities. This leads to:

Definition 3-3 Number of values scalar function $n = f(n_0, \ldots, n_{k-1})$

Let $n = f(n_0, \ldots, n_{k-1})$ be a scalar representation for a generic "number of values" in various array constructions. □

Unless otherwise specified, for the rest of this paper, we shall use $f(n_0, \ldots, n_{k-1}) = n_0$, and therefore $n = n_0$. Thus, n will represent the largest number of values for any of the parameters, according to the indexing convention in Definition 3-2. We shall also use n whenever all parameters have the same number of values, namely $n_0 = \ldots = n_{k-1}$.

One reason for introducing this value is that many constructions that have been developed assume the same number of values for each parameter, and the scalar value of n fulfils this role. Of course, this is not usually true in practice. Aside from conversion to a scalar value, another reason for introducing a separate variable is that some of the constructions that we shall use have additional constraints on n (for example, n may be required to be a prime number). These constraints can be incorporated into the function f.

We now turn to definitions and notations related to the concept of a test configuration, and assignments of values to parameters.

Definition 3-4 Indefinite value v_i assigned to a parameter i

For i, such that $0 \leq i < k$, there are n_i values for parameter i, which we can enumerate as $0, \ldots, n_i - 1$. Let v_i denote a value assigned to parameter i, where $0 \leq v_i < n_i$. □

The notation allows us to keep track, using the subscript, of which parameter has been assigned a value in situations when we shall refer to a partial set of parameter values.

Definition 3-5 Definite value assigned to a parameter

To refer to a specific definite value, the notation 1_3 indicates that parameter 3 takes the value 1. The subscript indicates the index of the value's associated parameter. □

Definition 3-6 Selection of a test configuration τ

To select a test configuration, exactly one value must be assigned to each of the k parameters, Therefore, a single test configuration is denoted $\tau = (v_0, \ldots, v_{k-1})$. □

In testing, we have a set of *possible* test configurations, and the selected subset of the test configurations that we will actually use for testing. The selection of the subset is based on a test coverage criterion. We now define notations for the set of *possible*, and *selected*, test configurations.

Definition 3-7 Set of all possible test configurations T

Let T denote the set of all possible test configurations, based on the values of k, and the n_i, namely $T = \{\tau = (v_0, \ldots, v_{k-1}) \mid 0 \leq v_i < n_i, 0 \leq i < k\}$. □

In other words, every parameter can be assigned any of its possible values.

The number of possible test configurations is $|T| = \prod_{i=0}^{k-1} n_i$. In the special case where all parameters have the same number of values n, we have $|T| = n^k$.

Definition 3-8 Selected test configurations S

Let S denote the set of test configurations selected for testing, where $S \subseteq T$. □

Here is a provisional test coverage criterion: our goal is to find $S \subseteq T$, such that S satisfies some test coverage property. Normally, an additional objective is to minimize $|S|$, the cardinality of the set of test configurations that satisfy the test coverage property. For the moment, we will leave the exact coverage property that S satisfies undefined until section 3.2.3.

We want to find the set of test configurations with minimal size that will satisfy our (currently undefined) test coverage criterion. Our goal is to define a test coverage criterion using a set of "test units" that need to be covered. We shall discuss what constitutes an interaction testing coverage unit in the next section.

3.2.2 Interaction Elements

To this point, all of the concepts defined and discussed in this section are generic in nature, in that they do not rely on any particular test selection strategy. In this section, we shall fill in the undefined test coverage criterion from the previous section, by introducing the concept of *interaction elements.*

The idea behind interaction elements is that they represent a "unit" of test coverage, in system component interaction testing. Other types of test "units," used in other contexts, (see, for example, [Bei90] or [Bi00]) are as follows:

- Control flow testing: statements, branches, flow graph nodes, flow graph edges, conditions.
- Data flow testing: definition-use pairs, IO-chains.

Construction of a test suite for the above methods is based on covering all, or as many as is practicable, of the test "units" by the test cases in the test suite.

In the context of system component interaction testing, the approach that we are taking is that there could be undesired or unforeseen interactions among system components when a particular set of values is assigned to the set of parameters. This should be detected when a test suite is run using the configuration defined by that set of parameter values.

The same is true for testing interactions among parameters contained within the same component. For example, in GUI testing (see section 2.4.2), there may be unforeseen interactions when different options are selected from a dialog. Similarly, when testing equivalence classes, there may be an unforeseen interaction among equivalence classes for different parameters.

We shall take the common view that an unwanted interaction is usually not caused by the particular values of the **entire** set of parameters, but by the values of only a (hopefully, small) **subset** of parameters. The philosophy of test coverage we shall adopt is to try to cover as many subsets of parameter values as possible, using the fewest number of configurations.

Furthermore, we shall distinguish the interactions by the size of the subset. We would like to identify subsets of size two, then three, and so on. The intention is that we will cover as many subsets of as large a size as possible, as permitted by the resources available for testing. The rationale here is that if an unwanted interaction is caused by two particular parameter values, it will also cause unwanted interactions with all parameter subsets of size three that include the two specific values that cause the problem. We can diagnose a problem more easily if interactions between two values are identified first. Then we can proceed to interactions among three specific parameter values, and so on.

Therefore, we shall limit ourselves to subsets of size $d \leq k$. These subsets will be referred to as interaction elements (or d-interaction elements, when referring to the cardinality of the subsets).

The aim is to reduce the number of test configurations to the point where the testing can be conducted with a feasible cost in time and money, and still have a good probability of detecting unwanted system interactions. While it would be ideal to have $d = k$, it will be shown in section 6.5 that the number of test configurations to meet the test coverage criterion will be proportional to n^d. In practice, constraints of time and money will usually dictate that a small value of d, such as $d = 2$ be used.

Before providing formal definitions, Figure 3-1 illustrates the approach we shall take. First, we look at test configurations: how they are constructed, and how many there are. We then discuss the two parts of an interaction: identifying the parameters that are involved, and the specific values that they have. In Figure 3-1, there are three parameters, and specific values A, B, and C have already been assigned to those parameters.

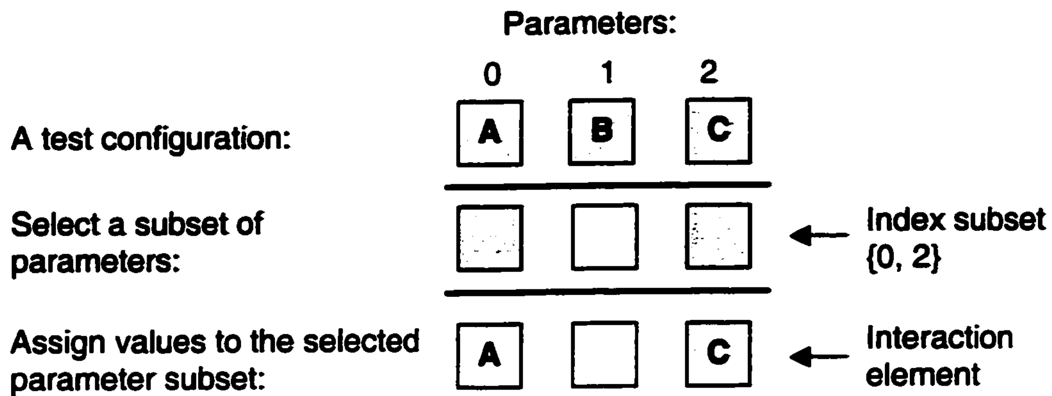


Figure 3-1: Construction of an interaction element

With this approach in mind, we shall turn to the definitions and notations of terms related to interaction elements.

Definition 3-9 Degree of interaction coverage *d*

The *interaction degree*, denoted d, is the size of the subsets of parameter values for which we wish to detect unwanted interactions. □

Definition 3-10 Index subset σ_d

Suppose that we have k parameters, indexed from 0 to $k - 1$. An index subset of size d, denoted σ_d, is $\sigma_d \subseteq \{0, \ldots, k-1\}$ where $|\sigma_d| = d$. □

In other words, we are selecting d parameters out of the set of the k parameters (see Figure 3-1). There is no particular order to the selection of parameters, so we shall use a set of d indices instead of an ordered d-tuple.

Definition 3-11 Set of all possible index subsets Φ_d

Let Φ_d denote the set of all possible index subsets of size d. Then:

$$\Phi_d = \{\sigma_d \mid \sigma_d \subseteq \{0, \ldots, k-1\}, |\sigma_d| = d\}. \square$$

The number of such index subsets is $|\Phi_d| = \binom{k}{d}$.

Definition 3-12 Interaction elements $\chi(\sigma_d, \tau)$

Suppose that $\tau = (v_1, ..., v_k)$ is a given test configuration in T, and σ_d is a given index subset of $\{0, ..., k - 1\}$. An *interaction element*, denoted by $\chi(\sigma_d, \tau)$ is defined as $\chi(\sigma_d, \tau) = \{v_j \mid v_j \in \tau, j \in \sigma_d\}$. The term *d-interaction element* will be used when it is important to emphasize the value of d, and the term *pair element* will be used for the special case of $d = 2$. □

In other words, we are selecting a subset, of size d, of the values for a given test configuration. In Figure 3-1, we isolate the interaction between value 'A' for parameter 0, and value 'C' for parameter 2.

Because we have chosen a subset of all the parameter indices, there are no repeated elements, and therefore a d-interaction element will not contain two values for the same parameter. This is a crucial test configuration property that we wish to preserve. The set notation is used to indicate that the ordering of the elements is not important. We can recover the parameter associations through the subscripts, as described in Definition 3-4 and Definition 3-5.

Suppose that we have five parameters, and we want to select the values for parameters 0, 2, and 3 from the test configuration $\tau = (4_0, 3_1, 2_2, 1_3, 0_4)$. For $d = 3$, and $\sigma_3 = \{0, 2, 3\}$, we have $\chi(\sigma_3, \tau) = \{4_0, 2_2, 1_3\}$. Note that the subscripts of the 3-interaction element match the index set.

Definition 3-13 Set of all interaction elements X_d

Let X_d denote the set of all possible d-interaction elements, for a system based on the values k, and n_i, $0 \leq i < k$. Then we have:

$$X_d = \{\{ v_j \mid j \in \sigma_d \} \mid 0 \leq v_j < n_j, \sigma_d \in \Phi_d\}. \quad \square$$

In other words, we take all possible index subsets of size d of the parameters. For each index subset, construct a set for every possible combination of d values for the associated parameters. The resulting collection is the set X_d.

While the definition of interaction elements is dependent on a specific test configuration τ, the set of all possible interaction elements X_d depends only on the number of parameters k, the number of values for each parameter n_i, and d. It is useful to keep the set X_d independent of any test configurations, and it is easier to determine the cardinality of X_d from this formulation.

Accordingly, we use the above definition, rather than an alternative, but equivalent, definition for X_d, namely:

$$X_d = \bigcup_{\tau \in T} \{ \chi(\sigma_d, \tau) \mid \sigma_d \in \Phi_d \}$$

In other words, we collect all interaction elements from all test configurations. The set union removes the duplicated elements that result from interaction elements being associated with more than one configuration.

Consider X_2, the set of all legal pair elements. The number of such elements is $|X_2| = \sum_{i=0}^{k-2} \sum_{j=i+1}^{k-1} n_i n_j$. In the special case when $n = n_i$ for all i in the range $0 \leq i < k$ (that is, all parameters have n values), we could have $\binom{k}{2}$ distinct subsets of parameters in a pair element. There are n^2 possible combinations of values for each subset of parameters. Therefore, we have $|X_2| = \binom{k}{2} n^2$.

For general values of d, the set X_d is the set of all legal d-interaction elements. The number of possible d-interaction elements is $|X_d| = \sum_{i_0=0}^{k-d} \ldots \sum_{i_{d-1}=i_{d-2}+1}^{k-1} \left(n_{i_0} \ldots n_{i_{d-1}}\right)$. In the special case when $n = n_i$ for all i in the range $0 \leq i < k$ (that is, all parameters have n values), $|X_d| = \binom{k}{d} n^d$.

3.2.3 The Interaction Test Coverage Problem

We have defined concepts related to system test configuration selection in section 3.2.1 and defined concepts related to interaction elements in section 3.2.2. We can now bring these concepts together to formally define the interaction test coverage problem.

Suppose that we consider a test configuration in an alternative form: as a set constructed from d-interaction elements. Within the k parameter values of any test configuration, there are $\binom{k}{d}$ possible subsets of size d of parameter values.

Figure 3-2 illustrates this relationship with an example. In Figure 3-2, A, B, and C represent specific values for a test configuration with three parameters. The figure shows the three pair elements contained within this test configuration.

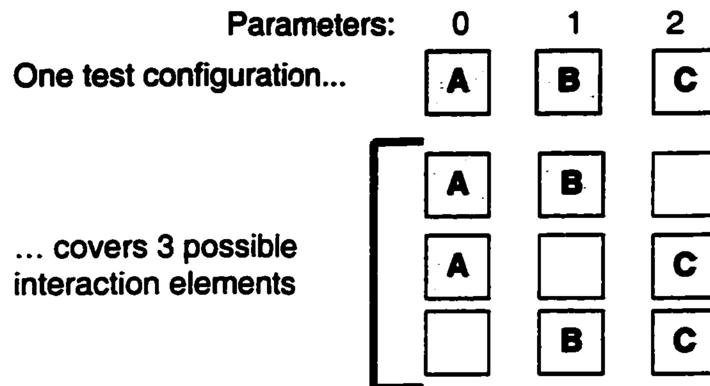


Figure 3-2: Interaction elements contained within a test configuration

This leads to:

Definition 3-14 Representation of a test configuration τ' as a set of d-interaction elements

Let $\tau = (v_0, \ldots, v_{k-1})$ be a specific test configuration. Instead of treating a configuration τ to be a set of k values, the test configuration τ can be represented (denoted τ') as a set of $\binom{k}{d}$ d-interaction elements $\tau' = \{ \chi(\sigma_d, \tau) \mid \sigma_d \in \Phi_d \}$. □

In other words, τ' is the set of interaction elements produced from all possible index subsets Φ_d for a single test configuration. Each interaction element must be an element of

X_d, as X_d contains all legal d-interaction elements over the entire range of values of the parameters.

For example, suppose there are three test parameters. Using our notation for definite assignment of values to parameters, the test configuration $\{0_0, 0_1, 0_2\}$ can also be represented as a set of pair interaction elements, namely $\{ \{0_0, 0_1\}, \{0_0, 0_2\}, \{0_1, 0_2\} \}$. In the latter form, we are considering this test configuration to consist of the three pairs of 0's produced by selecting parameters two at a time. In this form, a test configuration becomes a set of sets.

Clearly, each test configuration $\tau \in T$, can be represented as τ', a set of subsets of X_d, the set of all d-interaction elements. Therefore, the entire set of potential test configurations, T, is a collection of sets of subsets of X_d. We shall incorporate this significant property into our test coverage criterion.

Definition 3-15 Set of all possible test configurations T', expressed as interaction elements

Suppose that τ' is a test configuration expressed as a set of d-interaction elements based on $\tau = (v_0, \ldots, v_{k-1})$. We can then define T' the set of all possible test configurations expressed as d-interaction elements as $T' = \{ \{ \chi(\sigma_d, \tau) \mid \sigma_d \in \Phi_d \} \mid \tau \in T \}$. □

Each element of T' is a set of cardinality $\binom{k}{d}$, where each of those sets is, in turn, a set of cardinality d. There is a 1 to 1 correspondence between elements of T', and elements of T. Thus, the number of test configurations is $|T'| = \prod_{i=0}^{k-1} n_i$. In the special case where all parameters have the same number of values, $|T'| = n^k$.

At this point, we have defined X_d, the set of all possible interaction elements, which represents all possible d-way combinations of parameter values – the set of possible interactions. We have also expressed a test configuration as a set of interaction elements

– the set of interactions covered by each test configuration. We are now ready to define precisely a criterion for coverage of test interactions.

Definition 3-16 Set of selected test configurations S', expressed as interaction elements

Suppose that S is an arbitrary set of test configurations so that $S \subseteq T$. For each test configuration $\tau \in S$, let τ' be the equivalent test configuration expressed as a set of d-interaction elements $\tau' = \{ \chi(\sigma_d, \tau) \mid \sigma_d \in \Phi_d \}$. We can then define the set of all selected configurations expressed as d-interaction elements, denoted S' as $S' = \{ \{ \chi(\sigma_d, \tau) \mid \sigma_d \in \Phi_d \} \mid \tau \in S \}$□

Since $S \subseteq T$, we have $S' \subseteq T'$.

Definition 3-17 The interaction test coverage problem of degree *d*

We can now define the *interaction test coverage problem of degree d*: the goal is to find $S \subseteq T$, and thus $S' \subseteq T'$, such that every element in X_d belongs to at least one member of S'. An additional objective is to minimize $|S|$, the cardinality of the set of selected test configurations. □

In other words, we require that every d-interaction element be covered by a selected test configuration.

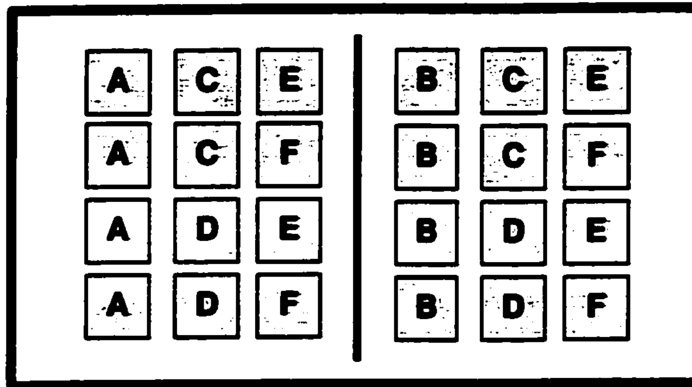
We would like the minimum number of test configurations that will cover all d-wise combinations among parameter values. Stevens and Moura [St98c] have shown that in many cases with $d = 2$, there is only a loose bound on the minimum number of configurations that can be achieved. Therefore, for all but the smallest problems, the practical objective will be to cover every interaction element with a set S, but the cardinality of S approximates the minimum possible size. It is this latter objective that we shall address in Chapter 6.

To highlight the number of interactions, we will use (for example) "3-interaction test coverage problem" to refer to the case when $d = 3$, and the set X_3 is the set of all 3-interaction elements.

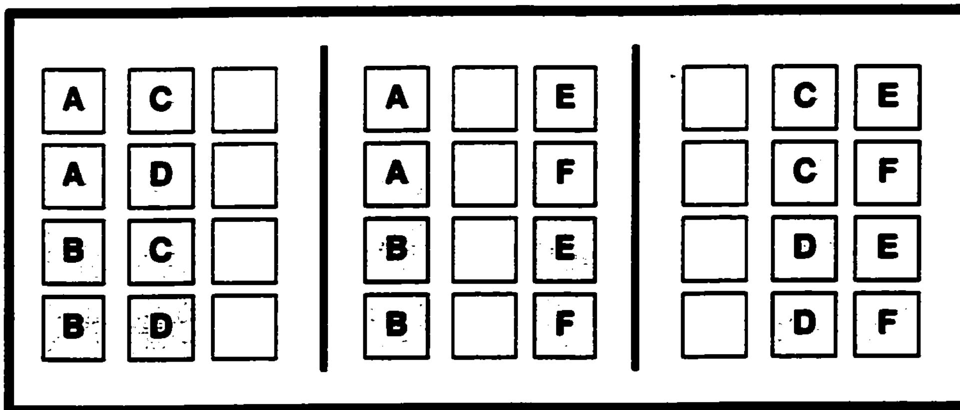
This definition corresponds to other types of test coverage, where the goal is to cover all (or as large a percentage as is possible) of some sort of test coverage unit, within the set of selected test configurations. We are using interaction elements as test units for system interaction testing, as one would use control flow branches or definition-use associations in other type of test coverage criteria. Since we are assuming that the parameters are independent, all of our potential test configurations are feasible in the sense that there is no equivalent to an infeasible path predicate in a data-flow oriented test method [Ur93]. Therefore, the test coverage criterion is to cover **all** interaction elements (subject to resource constraints).

Figure 3-3 shows an example where there are three parameters. Parameter 0, the left column, can take values 'A' or 'B'. Parameter 1, the middle column, can take values 'C' or 'D'. Parameter 2, the right column, can take values 'E' or 'F'. The 8 possible test configurations are shown, and then the 12 possible interaction elements. The goal is to cover the 12 interaction elements with a subset of the test configurations; the figure shows four test configurations that will achieve this coverage.

a) Goal: if these are the 8 possible test configurations for 3 parameters with 2 values each...



b) ...cover all 12 possible interaction elements...



c) ... using a subset of all test configurations.

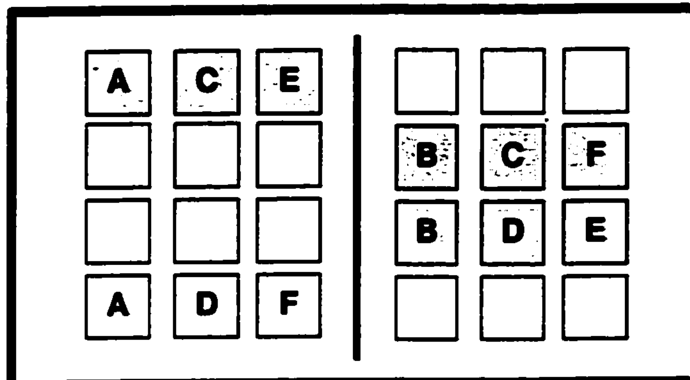


Figure 3-3: Coverage of Interaction Elements

3.3 Measurement of Interaction Test Coverage

3.3.1 Definition

In section 3.2.2, we noted that other traditional methods of test coverage have the concept of some sort of test unit. The objective is to cover all (or as many as is practicable) of the test units within a set of test configurations. Furthermore, coverage of a test suite can be measured as a percentage of the test units that are covered within the test suite.

The concept of interaction element can be used in this manner. For a given degree of interaction coverage d, we can count the number of d-interaction elements covered by a set of test configurations.

Definition 3-18 Set of interaction elements covered by a set of test configurations

Suppose that S is an arbitrary set of test configurations. Let $C_d(S)$ be the set of d-interaction elements covered by the set of selected test configurations S. Then:

$$C_d(S) = \bigcup_{\tau \in S} \{ \chi(\sigma_d, \tau) \mid \sigma_d \in \Phi_d \}. \square$$

This definition of $C_d(S)$ is slightly different from that for S' given in Definition 3-16. The set S' represents each test configuration as a set of interaction elements, and so it is a "set of sets of sets". The set $C_d(S)$ collapses this one level, and uses the set union to remove duplicate interaction elements. The result is a set of interaction elements, which are in turn, sets. The cardinality of $C_d(S)$ is an important component of our coverage measure, and therefore it is essential that the duplicates be eliminated.

We are now ready to define a measurement of interaction test coverage.

Definition 3-19 Measurement of Interaction Test Coverage of specified degree

Let T be the set of all possible test configurations, and suppose that S is an arbitrary set of test configurations with $S \subseteq T$. Let X_d be the set of all possible d-interaction elements, and $C_d(S)$ be the set of d-interaction elements covered by S.

Then the *interaction coverage of degree d* is $c_d(S) = \frac{|C_d(S)|}{|X_d|}$. $\square$

The coverage c_d will be a value such that $0 \le c_d(S) \le 1$, where 1 represents coverage of all interaction elements. The value can also be expressed as a percentage.

Note that the coverage value $c_d(S)$ is measured with respect to a specified degree of interaction coverage d, and with respect to a specific selected set of test configurations S. The coverage value will change over different values of d, for the same set of test configurations.

3.3.2 Example of Coverage Measurement

Suppose that we have four parameters, and three values for each parameter. Then there are $3^4 = 81$ possible test configurations. For degree 2 interaction coverage, there are $\binom{4}{2}3^2 = 54$ possible interaction elements. For degree 3 interaction coverage, there are $\binom{4}{3}3^3 = 108$ interaction elements.

Suppose that we arbitrarily choose the test configuration $\{0_0, 0_1, 0_2, 0_3\}$. A single test configuration covers $\binom{4}{2} = 6$ interaction elements of degree 2. Specifically, the pair elements $\{0_0, 0_1\}$, $\{0_0, 0_2\}$, $\{0_0, 0_3\}$, $\{0_1, 0_2\}$, $\{0_1, 0_3\}$, and $\{0_2, 0_3\}$ are covered. With this single test configuration, we have degree 2 coverage $c_2 = \frac{6}{54} \times 100\% = 11.1\%$.

For degree 3, each test configuration covers $\binom{4}{3} = 4$ interaction elements of degree 3. Specifically, the 3-interaction elements $\{0_0, 0_1, 0_2\}$, $\{0_0, 0_1, 0_3\}$, $\{0_0, 0_2, 0_3\}$, and $\{0_1, 0_2, 0_3\}$ are covered, and therefore the single test configuration has degree three coverage of $c_3 = \frac{4}{108} \times 100\% = 3.7\%$.

Note that for degree four coverage, one test configuration equates to one interaction element, so that we can also say that we have $c_4 = \frac{1}{81} \times 100\% = 1.2\%$.

In adding successive test configurations, we would ideally like to be able to cover a set of interaction elements that has no overlap with the ones that are already covered. This

would result in achieving the greatest coverage with the fewest configurations. However, this may not always be possible.

Suppose that the next test configuration that is chosen is $\{0_0, 1_1, 2_2, 0_3\}$. This covers additional pair elements $\{0_0, 1_1\}$, $\{0_0, 2_2\}$, $\{1_1, 2_2\}$, $\{1_1, 0_3\}$, and $\{2_2, 0_3\}$. However, $\{0_0, 0_3\}$ has already been covered. Therefore, the cumulative degree 2 coverage is $c_2 = \frac{11}{54} \times 100\% = 20.3\%$.

For degree 3 coverage, this test configuration covers four new 3-interaction elements: $\{0_0, 1_1, 2_2\}$, $\{0_0, 1_1, 0_3\}$, $\{0_0, 2_2, 0_3\}$ and $\{1_1, 2_2, 0_3\}$. Therefore, the cumulative degree 3 coverage is $c_3 = \frac{8}{108} \times 100\% = 7.4\%$.

Note that this configuration covers the maximum possible number of additional interaction elements of degree 3, but not for degree 2. If our goal is to achieve degree two coverage with the fewest possible configurations, it might be better to choose a different test configuration at this point. It should be reiterated that it is not always possible to choose a test configuration that covers the maximum possible number of interaction elements of a particular degree. If it were, then the minimum number of test configurations to achieve 100% coverage of degree d would always be n^d. This is not the case. For example, if there are four parameters and six values for each, it has been shown [Hal86] that one cannot find a set of 36 test cases to cover all pair-wise interactions. The minimum number is 37, as shown in [St98b].

We present an example with four parameters, and three values for each parameter. Suppose that the set of test configurations in Table 3-1 has been selected:

Config.	Interaction Elements covered
$\{0_0, 0_1, 0_2, 0_3\}$	$\{0_0, 0_1\}, \{0_0, 0_2\}, \{0_0, 0_3\}, \{0_1, 0_2\}, \{0_1, 0_3\}, \{0_2, 0_3\}$
$\{0_0, 1_1, 1_2, 1_3\}$	$\{0_0, 1_1\}, \{0_0, 1_2\}, \{0_0, 1_3\}, \{1_1, 1_2\}, \{1_1, 1_3\}, \{1_2, 1_3\}$
$\{0_0, 2_1, 2_2, 2_3\}$	$\{0_0, 2_1\}, \{0_0, 2_2\}, \{0_0, 2_3\}, \{2_1, 2_2\}, \{2_1, 2_3\}, \{2_2, 2_3\}$
$\{1_0, 0_1, 1_2, 2_3\}$	$\{1_0, 0_1\}, \{1_0, 1_2\}, \{1_0, 2_3\}, \{0_1, 1_2\}, \{0_1, 2_3\}, \{1_2, 2_3\}$
$\{1_0, 1_1, 2_2, 0_3\}$	$\{1_0, 1_1\}, \{1_0, 2_2\}, \{1_0, 0_3\}, \{1_1, 2_2\}, \{1_1, 0_2\}, \{2_2, 0_3\}$
$\{1_0, 2_1, 0_2, 1_3\}$	$\{1_0, 2_1\}, \{1_0, 0_2\}, \{1_0, 1_3\}, \{2_1, 0_2\}, \{2_1, 1_3\}, \{0_2, 1_3\}$
$\{2_0, 0_1, 2_2, 1_3\}$	$\{2_0, 0_1\}, \{2_0, 2_2\}, \{2_0, 1_3\}, \{0_1, 2_2\}, \{0_1, 1_3\}, \{2_2, 1_3\}$
$\{2_0, 1_1, 0_2, 2_3\}$	$\{2_0, 1_1\}, \{2_0, 0_2\}, \{2_0, 2_3\}, \{1_1, 0_2\}, \{1_1, 2_2\}, \{0_2, 2_3\}$
$\{2_0, 2_1, 1_2, 0_3\}$	$\{2_0, 2_1\}, \{2_0, 1_2\}, \{2_0, 0_3\}, \{2_1, 1_2\}, \{2_1, 0_3\}, \{1_2, 0_3\}$

Table 3-1: Set of selected test configurations

In this case, there are $\binom{4}{2}3^2 = 54$ possible interaction elements of degree 2. There are also 54 interaction elements covered by this set of test configurations; none of them is duplicated. Therefore, this set of test configurations has c_2 coverage of 100%.

3.4 Comparison with well-known combinatorial optimization problems

In this section, we compare the interaction test coverage problem with combinatorial optimization problems that are based on graphs, sets, and integer programming. It should be apparent from the preceding section that there will be a greater correspondence with the set and integer programming problems, versus a formulation based on graphs.

We shall see that the interaction test coverage problem can be related to the set cover problem, and the {0, 1} integer programming problem, both of which are NP-complete [LR79][Cr].

For the purposes of investigating whether the interaction test coverage problem is NP-complete, it is necessary to formulate the problem as a decision problem:

Definition 3-20 The interaction test coverage problem of degree *d* (as a decision problem)

- Let T be the set of all possible test configurations, and T' be the corresponding set of all test configurations expressed as sets of d-interaction elements. Let S be a set of arbitrary selected test configurations with $S \subseteq T$, and S' be the corresponding set of selected configurations expressed as sets of d-interaction elements such that we have $S' \subseteq T'$. Is there a set S with $|S| \leq k$, for given integer k, such that every member of X_d, the set of all possible interaction elements of degree d, belongs to at least one member of S'? □

3.4.1 Comparison with the Minimum Set Cover problem

We now compare the interaction test coverage problem with the general case of the minimum set cover problem [Pa98] defined as follows:

Definition 3-21 The minimum set cover problem

Let A be a finite set. Let B be a collection of subsets of A. Let C be a subset $C \subseteq B$ such that every element in A belongs to at least one member of C. The objective is to minimize $|C|$, the cardinality of the set cover. □

The general minimum set cover problem is NP-complete [LR79].

We can now construct a mapping of the interaction test coverage problem to the minimum set cover problem. The set X_d, the set of all interaction elements of degree d, maps on to the set A in the minimum set cover problem. The set T', the set of test configurations expressed as interaction elements maps onto the set B. The set S', the set of selected test configurations expressed as interaction elements maps onto the set C.

The interaction test coverage problem is an instance of the general set cover problem, but the converse is not true, as there are some additional considerations. The general set cover problem does not reduce to the interaction test coverage problem because the following properties are also present in the interaction test coverage problem:

- All sets in T' have identical cardinality.

In fact, every selected test configuration (that is, every element of T') contains exactly $\binom{k}{d}$ d-interaction elements. This is because every test configuration contains k elements. The subsets that are used to comprise the set cover are always of identical cardinality for an interaction test coverage problem, which is not the case in a general set cover problem.

- The number of occurrences in T' of any element of X_d is at least $n_d \times n_{d+1} \times \ldots \times n_{k-1}$.

  Each d-interaction element occurs exactly n^{k-d} times in T', assuming there is the same number of values for each parameter. When there are differing numbers of values for each parameter, the number of occurrences of any d-interaction element will vary somewhat, but will be at least $n_d \times n_{d+1} \times \ldots \times n_{k-1}$.

  This means that for $d < k$, any particular element required for the set cover will occur in a "reasonable" number of subsets. There will not be the situation that could happen in a general set cover problem where there is one elusive element that exists only in one subset – and therefore requires examining all subsets to find it. (Note that $d = k$ is a degenerate case where we would select the entire set T').

These additional properties of the interaction test coverage problem mean that we cannot reduce a general set cover problem to an interaction test coverage problem, and cannot directly use the set cover problem to prove the NP-completeness of the interaction test coverage problem.

3.4.2 Formulation as a constraint-based problem

Another re-formulation of the interaction test coverage problem is as a constraint-based problem. The result will become a {0, 1} integer programming problem.

Our objective in the interaction test coverage problem is to cover all of the interaction elements using a subset of all possible configurations. That is, given the set X_d of all interaction elements, and the set of all possible test configurations T, select $S \subseteq T$ as the

set of selected test configurations. When each element of S is expressed as a set of interaction elements, the set S' is formed as the collection of all covered interaction elements. Then every element of X_d must be in S'.

Suppose that we construct an enumeration of the elements of T. The number of possible test configurations is $|T| = \prod_{i=0}^{k-1} n_i$. Let $z = |T|$. Furthermore, we will also construct an enumeration of the set of all possible interaction elements. The number of interaction elements is $|X_d| = \sum_{i_0=0}^{k-d} \cdots \sum_{i_{d-1}=i_{d-2}+1}^{k-1} \left(n_{i_0} \ldots n_{i_{d-1}}\right)$. Let $y = |X_d|$. In the special case where all parameters have the same number of values n, $z = n^k$ and $y = \binom{k}{d} n^d$.

Set up variables $\{x_0, \ldots, x_{z-1}\}$, where the value of $x_j = 1$ if configuration number j is selected (i.e., contained in S), and 0 if configuration number j is not selected. Since the number of test configurations grows exponentially in the number of parameters, this will be a large set of variables for all but the smallest of test coverage problems.

Now, for each interaction element, determine the set of configurations that cover that interaction element. These will be all configurations where the d parameters in the index set for the interaction element are fixed and the remaining $k - d$ parameters can take any possible value. In the case where there are n values for each parameter, this means that there are n^{k-d} configurations that would cover this interaction element. Define an array of coefficients A such that $a_{ij} = 1$ if interaction element number i is covered by configuration number j, and 0 otherwise. For any i, there will be $k - d$ non-zero values of a_{ij}.

For each of the n^{k-d} configurations, we have a set of n^{k-d} variables representing whether or not the configuration is selected. Therefore, the sum of those variables represents the number of times the particular interaction element is covered. This sum is $\sum_{j=0}^{z-1} a_{ij} x_j$.

To meet our coverage goal, we require that the sum of those variables is greater than or equal to one, so that the interaction element is covered by at least one test configuration.

This represents one constraint on our set of selected test configuration. Therefore, the set of y interaction elements will produce a set of y constraints of the form $\sum_{j=0}^{z-1} a_{ij}x_j \geq 1$, where $0 \leq i < y$.

Additionally, we would like to minimize the total number of configurations that are selected. That is, we would like $\sum_{j=0}^{z-1} x_j$ to be minimized.

Put together, this forms a {0, 1} integer-programming problem: minimize $\sum_{j=0}^{z-1} x_j$ subject to the constraints $\sum_{j=0}^{z-1} a_{ij}x_j \geq 1$, where $0 \leq i < y$.

At this point, a comment on the restriction of the values x_j to 0 or 1 is in order. Clearly, the values must be restricted to non-negative integers, as a test configuration cannot be "partially selected" (a fractional value) or "negatively selected". However, suppose that we relax the restriction on the values of x_j to be non-negative integers instead.

Furthermore, suppose that in a minimized sum $\sum_{j=0}^{z-1} x_j$, there is a value $x_q > 1$. Since all of the constraints are of the form $\sum_{j=0}^{z-1} a_{ij}x_j \geq 1$, where $a_{ij} \in \{0, 1\}$, we could replace the value of x_q by 1, and the constraint would still be satisfied. Furthermore, since all of the x_j are non-negative integers, the sum $\sum_{j=0}^{z-1} x_j$ would be reduced by the amount $x_q - 1$, which contradicts the assumption that the sum was minimized. Therefore, in the minimized sum, all of the $x_j \in \{0, 1\}$.

The result is that the restriction $x_j \in \{0, 1\}$ does not exclude any potential minimal solutions. The restriction may be useful in practice, as we can provide an integer-programming solver with the information that the variables are binary in nature. This

allows the integer-programming solver to discard potential solutions more quickly. (Note that either of these are still much more difficult to solve as compared with a linear programming problem.)

Here is an example. Suppose that there are three parameters, and there are two possible values for each parameter. There are $2^3 = 8$ possible test configurations, and $\binom{3}{2}2^2 = 12$ interaction elements to be covered. We define a set of variables $\{x_0, \ldots, x_7\}$ such that $x_j \in \{0, 1\}$. We will have $x_j = 1$ if configuration j is selected, and 0 otherwise. Figure 3-4 illustrates the resulting {0,1} integer programming problem.

In Figure 3-4, the sets of test configurations are listed in the column headings, and the sets of interaction elements are listed in the row headings.

	{0,0,0}	{0,0,1}	{0,1,0}	{0,1,1}	{1,0,0}	{1,0,1}	{1,1,0}	{1,1,1}	
$\{0_0,0_1\}$	x_0	$+x_1$							≥ 1
$\{0_0,0_2\}$	x_0		$+x_2$						≥ 1
$\{0_1,0_2\}$	x_0				$+x_4$				≥ 1
$\{0_0,1_1\}$			x_2	$+x_3$					≥ 1
$\{0_0,1_2\}$		x_1		$+x_3$					≥ 1
$\{0_1,1_2\}$		x_1				$+x_5$			≥ 1
$\{1_0,0_1\}$					x_4	$+x_5$			≥ 1
$\{1_0,0_2\}$					x_4		$+x_6$		≥ 1
$\{1_1,0_2\}$			x_2				$+x_6$		≥ 1
$\{1_0,1_1\}$							x_6	$+x_7$	≥ 1
$\{1_0,1_2\}$						x_5		$+x_7$	≥ 1
$\{1_1,1_2\}$				x_3				$+x_7$	≥ 1

Figure 3-4: Formulation of a test coverage problem as an integer program

If we minimize $x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7$, subject to $x_0 + x_1 \geq 1$, $x_0 + x_2 \geq 1$, $x_0 + x_4 \geq 1$, $x_2 + x_3 \geq 1$, $x_1 + x_3 \geq 1$, $x_1 + x_5 \geq 1$, $x_4 + x_5 \geq 1$, $x_4 + x_6 \geq 1$, $x_2 + x_6 \geq 1$, $x_6 + x_7 \geq 1$, $x_5 + x_7 \geq 1$, $x_3 + x_7 \geq 1$, we will have obtained the set of test configurations of minimal cardinality that covers all interaction elements.

For experimental purposes, a mixed integer and linear programming solver called "lp_solve" [Ber], written by Michel Berkelaar, was used. Lp_solve is available as freeware from Eindhoven University. The input file format is straightforward, given the above discussion:

```
min: x0 + x1 + x2 + x3 + x4 + x5 + x6 + x7;
x0 + x1 >= 1;
x0 + x2 >= 1;
x0 + x4 >= 1;
x2 + x3 >= 1;
x1 + x3 >= 1;
x1 + x5 >= 1;
x4 + x5 >= 1;
x4 + x6 >= 1;
x2 + x6 >= 1;
x6 + x7 >= 1;
x5 + x7 >= 1;
x3 + x7 >= 1;
int x0, x1, x2, x3, x4, x5, x6, x7;
```

The first line represents the objective function, and indicates that it should be minimized. The constraints are listed, and the final line indicates that variables x0 through x7 should be restricted to integers. An implied constraint is that all variables are greater than or equal to zero.

The output produced by lp_solve for the above input file is:

```
CPU Time for parsing input: 0.01s (0.01s total since program start)
CPU Time for solving: 0s (0.01s total since program start)
Value of objective function: 4
x0                    1
x1                    0
x2                    0
x3                    1
x4                    0
x5                    1
x6                    1
x7                    0
```

The value of the objective function is 4, which is the sum of x0 through x7. This indicates that four test configurations are sufficient to provide coverage of all interaction elements. The specific values of the solution variables indicate that the test configurations $\{0, 0, 0\}$, $\{0, 1, 1\}$, $\{1, 0, 1\}$, and $\{1, 1, 0\}$ should be selected. (The configurations can be determined quickly by representing the variable number in binary: for example, x5 represents $\{1, 0, 1\}$.)

By formulating the interaction test coverage problem as a {0, 1} integer programming problem, solving the resulting integer program exactly will produce the best possible solution.

However, the difficulty is that the number of variables required grows exponentially. If there is the same number of values, n, for each parameter, we have the following results:

- The row dimension will be $\binom{k}{d}n^d$, which is the number of interaction elements. This is somewhat manageable, since the power in the exponent is only d.
- The column dimension will be n^k, which is the number of configurations. This will be extremely large in most cases.
- The number of non-zero a_{ij} per row will be n^{k-d}
- The number of non-zero a_{ij} will be $\binom{k}{d} n^k$

To get an idea of how quickly this becomes infeasible to calculate in practice, here are some experimental results using the lp_solve tool:

# parms	# values per parm	# constraints	# variables	Result: # configs	Run time (s)
3	2	12	8	4	<0.01
4	2	24	16	5	0.01
5	2	40	32	6	0.70
6	2	60	64	6	16.57
7	2	84	128	6	441.21
4	3	54	81	9	0.08
5	3	90	243	13*	*

Table 3-2: Results of running lp_solve on test configuration problems

The * in Table 3-2 indicates the best possible solution found at the point when the process was killed after running for about 6 ½ hours. Some results provided by other users of lp_solve, (included as documentation in the distribution) indicate lp_solve will

have difficulties for problems that have more than about 100 **integer** variables. The above results support this statement.

There are distinctions between the integer programming problem and the interaction test coverage problem that are similar to what we observed for the minimum set cover problem. That is, the interaction test coverage problem will result in a set of constraints where each variable appears in at least $n_d \times n_{d+1} \times \dots \times n_{k-1}$ constraints. We can also guarantee that there is a feasible minimum solution to the integer-programming problem constructed out of a test configuration problem. There has to be at least one feasible solution since we are guaranteed to cover every interaction element by selecting every possible test configuration. Of course, the minimum ought to be much less than that, but we have shown the existence of at least one solution, and therefore a minimum has to exist. A general {0,1} integer program does not necessarily have any feasible solution. Thus, we expect the reduction of the {0,1} integer programming problem to the interaction test coverage problem to be unlikely.

3.5 Chapter Summary

In this chapter, we have formally defined the interaction test coverage problem, and a metric for measuring interaction test coverage, using the concept of interaction element. The goal of the interaction test coverage problem is to cover all possible interaction elements, using the smallest possible subset of the set of all test configurations. Furthermore, we have defined a coverage metric to be the percentage of the set of all possible interaction elements that is covered by a selected set of test configurations.

To try to show NP-completeness of the interaction test coverage problem of degree d, we tried to reduce the minimum set cover problem to the interaction test coverage problem,. Moreover, even though the interaction test coverage problem can be formulated as either a minimum set cover problem or a {0, 1} integer programming problem, both of which are NP-complete, we have not reduced either problem to the interaction test coverage problem.

The general cases of the minimum set cover problém or the $\{0, 1\}$ integer-programming problem cannot be directly formulated as an interaction test coverage problem. This prevents us from using these related problems to show that the interaction test coverage problem is NP-complete. Thus, the NP-completeness of the interaction test coverage problem of degree d remains open.

In the next chapter, we turn to the field of statistical experimental design, to investigate methods of generating a small set of test configurations that achieves coverage of all interaction elements of degree 2.

Chapter 4 Statistical Experimental Design

4.1 Chapter Introduction

In this chapter, we introduce the topic of statistical design of experiments, with the aim of determining how to apply the approach to software testing. We outline the basic concepts of a statistical experiment: the objectives we are trying to achieve with the experiment, and the inputs to the experiment. This is described in the context of a traditional application of this approach, such as testing interactions among fertilizers and herbicides for agriculture, or testing for undesired side effects among prescription drugs.

We begin by looking at the principles of experimental design for general situations. We then review the use of orthogonal arrays, which is a standard technique used for constructing experiments in general. In particular, we present an algorithm for constructing orthogonal arrays, and review previous work in this area.

The discussion in this chapter would apply to any generic experimental design. We shall discuss the specifics using this approach for software testing in Chapter 5. In that chapter, we will see what improvements to the traditional approach can be made, based on the specific properties that arise in the software testing context.

4.2 Principles of experimental design

A general statistical experimental design is constructed on the assumption that an experiment has several real-valued inputs, and at least one real-valued output. It is assumed that any particular input parameter (called a *factor* in statistical literature) can be set independently to any legitimate value. The objective is to be able to detect the effect of individual parameters on the output, and to detect interaction effects among the parameters. The latter are particularly critical, for example, when testing potential drug treatments to avoid harmful side effects.

In developing these methods, considerable attention is paid to the identification of *factors* and *levels* for the experiments. The *factors* are the parameters to include, and *levels* are

the values of the factors to use in the experiment. Since the parameters can typically have any real value, a small number of representative values are selected as "levels" that are the discrete inputs to use in the experiment. The levels must be chosen to provide a suitable range of inputs, and to cause output effects that are amenable to result analysis.

A single experiment then consists of a combination of discrete factor levels, one per parameter. The complete design consists of a set of such combinations. Performing the experiments in the design produces a set of real-valued results that must then be analyzed. Therefore, the design must be constructed so that the effect of any individual factor can be identified as positive, negative, or neutral. Furthermore, if desired, the design can be set up so that interactions of various combinations of factors can be analyzed as positive, negative, or neutral. Different levels of coverage are possible based on the *degree* of the interactions, namely, how many factors are included in the combinations of interest.

Another important element of experimental design is to set up the analysis of the real-valued experiment results, so that the significance of each factor in the experiment can be determined with a known level of confidence. To ensure that the influence of different factors can be determined from the real-valued result, there is a "balance" property. For example, to determine how the levels of parameter 0 affect the result, it is necessary that each of the levels be contained in experiments with the complete range of levels of parameter 1. Furthermore, each level of parameter 0 must be in the same number of experiments as each level of parameter 1. The result is that the effect of the variation in levels of parameter 1 can be eliminated, and the effect of parameter 0 can be extracted. This leads to a balanced design where each combination of factor levels is used in the same number of experiments.

4.3 Orthogonal arrays

The method of orthogonal arrays is an experimental design construction technique from the literature of statistics. In turn, construction of such arrays depends on the theory of combinatorics.

An orthogonal array is a balanced two-way classification scheme used to construct balanced experiments when it is not practical to test all possible combinations. The size and shape of the array depend on the number of parameters and values in the experiment.

Orthogonal arrays are related to *combinatorial designs*. A *design*, represented by a set of orthogonal Latin squares (the precise definition will appear in section 4.4.2), can be easily converted into the form of an orthogonal array. The construction and properties of orthogonal Latin squares derive from both algebraic and combinatorial theories. Gilbert [Gi76] includes an introduction to the algebraic construction of orthogonal Latin squares. Hall [Hal86] covers the combinatorial theory and properties of orthogonal Latin squares. While sets of orthogonal Latin squares are known to exist in some cases, and not in others, there are many open problems relating to their existence or non-existence. A catalogue of results known as of 1996 is contained in [Col96], with subsequent on-line updates in [Din].

Definition 4-1 Orthogonal array $O(\rho, k, n, d)$

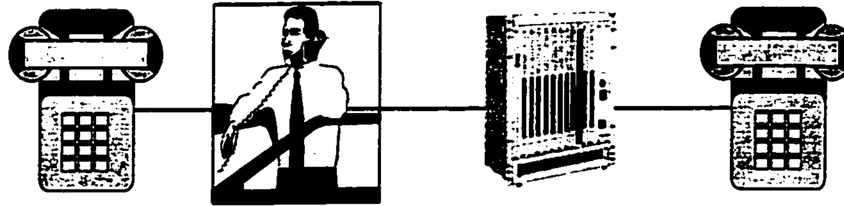
An orthogonal array is denoted by $O(\rho, k, n, d)$, where:

- ρ is the number of rows in the array. The k-tuple forming each row represents a single test configuration, and thus ρ represents the number of test configurations.
- k is the number of columns, representing the number of parameters.
- The entries in the array are the values 0, ..., $n - 1$, where $n = f(n_0, \ldots, n_{k-1})$. Typically, this means that each parameter would have (up to) n values.
- d is the *strength* of the array (see below).

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

balance property defines the orthogonal array. This property allows the effect of a single parameter to be detected, as well as interactions among (up to) d parameters. □

To relate this to system testing, suppose that we have the scenario in Figure 4-1:



Calling phone:	Call type:	Switch market:	Called phone:
Regular	Local	Canada	Regular
Cellular	Long distance	US	Cellular
Pay phone	Toll free	Mexico	Pager

Figure 4-1: A system test scenario

The array $O(9, 4, 3, 2)$, shown in Table 4-1, can be used to provide a set of test configurations that achieve pair-wise parameter coverage for the situation in Figure 4-1.

Configuration	Parameter			
	0	1	2	3
0	0	0	0	0
1	0	1	1	1
2	0	2	2	2
3	1	0	1	2
4	1	1	2	0
5	1	2	0	1
6	2	0	2	1
7	2	1	0	2
8	2	2	1	0

Table 4-1: The orthogonal array $O(9, 4, 3, 2)$

In Table 4-1, the columns represent the parameters, and the rows represent the test configurations. The entry at each location represents which value to use for each parameter for that test configuration.

Suppose that we number the parameter values as shown in Table 4-2:

Calling Phone	Call Type	Call Type	Called phone:
0 = Regular	0 = Local	0 = Canada	0 = Regular
1 = Cellular	1 = Long Distance	1 = US	1 = Cellular
2 = Pay phone	2 = Toll Free	2 = Mexico	2 = Pager

Table 4-2: Enumeration of parameter values

Using the enumeration in Table 4-2, Table 4-3 shows the actual test configurations for the scenario in Figure 4-1 to achieve pair-wise parameter coverage.

Configuration number	Calling phone	Call type	Switch Market	Called phone
0	Regular	Local	Canada	Regular
1	Regular	Long distance	US	Cellular
2	Regular	Toll free	Mexico	Pager
3	Cellular	Local	US	Pager
4	Cellular	Long distance	Mexico	Regular
5	Cellular	Toll free	Canada	Cellular
6	Pay phone	Local	Mexico	Cellular
7	Pay phone	Long distance	Canada	Pager
8	Pay phone	Toll free	US	Regular

Table 4-3: Test configurations for the system test scenario

If you select any two columns in Table 4-3, each of the parameter values for one column has a configuration where it is tested against each one of the other parameter's values. Thus, all pair-wise combinations have been covered, in only nine configurations out of the 81 that are possible. Clearly, pair-wise coverage requires at least $3 \times 3 = 9$ configurations. In this case, we have achieved the fewest possible configurations to meet the coverage criteria of all pair-wise parameter values, as the orthogonal array has strength 2.

4.3.1 Properties of Orthogonal arrays

There are several properties of orthogonal arrays that are useful to highlight at this point.

In the design of an experiment, several properties should be preserved. We should expect that since the parameters are independent, we could order them arbitrarily. Likewise, we should be able to arbitrarily select an order for the enumeration of each parameter's values. In addition, there should be no particular significance to the order of the set of experiments.

Therefore, in terms of an orthogonal array, these properties mean that we should be able to arbitrarily interchange the rows, representing the order of configurations, and columns, representing the order of parameters, of the array. As well, once we choose the order of the discrete values for each parameter, we can assign any permutation of the numbers 0 through $n - 1$ to enumerate the values.

The first property is that we can interchange the rows of an orthogonal array and still preserve all the properties of the array. In Definition 4-1, we have that an orthogonal array has strength d if we can select any d columns; each of the n^d possible rows appears the same number (>0) of times. Clearly, interchanging two rows will not alter this property.

Likewise, we can also interchange the columns in the array. This property corresponds to reordering the parameters in the experiment. If the parameters are independent of each other, their order should be irrelevant. The strength attribute is preserved because there is a 1 to 1 correspondence between the previous and new ordering of the parameters. This leads to a 1 to 1 correspondence between the k-tuples formed by each row.

Another property is that we can take the set of entries in the array (the values 0 through n - 1), and apply a permutation on this set. For example, if the entries in the array are from the set {0, 1, 2}, we can replace all the 0 entries by 2, and all the 2 entries by 0, and we will still have an orthogonal array. This property corresponds to reordering the values in the experiment. Because the values are discrete, we should be able to change their order arbitrarily.

One last useful property is that we can delete a column from an orthogonal array, and the resulting array is still orthogonal. This follows from the fact that we are checking the

strength by selecting columns in groups of d columns at a time. If we remove a column, all the other choices of a set of d columns at a time are unaffected; there are simply fewer choices. The effect of this property means that if we have k parameters, we need to find an orthogonal array with at least k columns, instead of exactly k columns. Removing a column does not affect the number of configurations. If we choose an array with additional columns over what it required, it may also result in a larger number of configurations as compared with a smaller array.

However, this property lets us use two strategies. The first is that for an arbitrary value of k, if we know of the existence of an orthogonal array with additional columns but the same number of configurations, we can use that array and then simply delete the unneeded columns. Second, if we cannot find an array of the appropriate size, we can use a larger orthogonal array for more parameters (and possibly more values), at the expense of additional configurations.

4.4 Orthogonal array construction

How is an orthogonal array generated? A sufficient, but not necessary, approach is that the orthogonal array is generated from a set of orthogonal Latin squares.

4.4.1 Latin squares

Definition 4-2 Latin square

Let Σ be a set of n alphabet symbols. A *Latin square* is an $n \times n$ matrix $[a_{ij}]$, $a_{ij} \in \Sigma$ such that if we choose $0 \leq i, j < n$, then $a_{ij} \neq a_{ic}$ and $a_{ij} \neq a_{rj}$, for all $0 \leq r, c < n$ where $j \neq c$ and $i \neq r$. In other words, each member of the alphabet set appears exactly once in each row and column. □

We will normally use the values $\{0, \ldots, n-1\}$ as the alphabet set, as shown in Figure 4-2.

$$\begin{matrix} 0 & 1 & 2 \\ 1 & 2 & 0 \\ 2 & \boxed{0} & 1 \end{matrix}$$

Figure 4-2: A 3 × 3 Latin square

The alphabet set Σ will correspond to the range of values that each parameter can take. There is an implicit assumption that each parameter has n values. In practice, this is not always true. This is the reason that the function $n = f(n_0, \ldots, n_{k-1})$ was introduced in section 3.2.1. For the moment, we shall use $f(n_0, \ldots, n_{k-1}) = n_0$, the maximum number of values.

A single $n \times n$ Latin square will produce the orthogonal array $O(n^2, 3, n, 2)$; it can cover two-way interactions for three parameters, each with n values, and produces n^2 configurations. Each entry in the Latin square represents the values for a single test configuration. Parameter 0 takes the value of the row index of the entry. Parameter 1 takes the value of the column index of the entry. The remaining parameter takes the value of the matrix entry. For example, the highlighted matrix entry in Figure 4-2 represents the test configuration $\{2_0, 1_1, 0_2\}$: row 2, column 1, and entry 0.

4.4.2 Orthogonal Latin squares

To handle more than three parameters, we need a set of orthogonal Latin squares.

Definition 4-3 Orthogonal Latin squares

> A set of k squares is said to be *orthogonal*, if the combined matrix formed by ordered k-tuples of the individual matrix entries has no repeated elements. Suppose we have two matrices $[a_{ij}]$ and $[b_{ij}]$. The combined matrix is $[c_{ij}]$, where the elements are $c_{ij} = (a_{ij}, b_{ij})$. If $c_{ij} \neq c_{rq}$ whenever $r \neq i$ and $q \neq j$, then the squares $[a_{ij}]$ and $[b_{ij}]$ are orthogonal. □

If there are k system parameters, then $k - 2$ orthogonal Latin squares are required to produce $O(n^2, k, n, 2)$. With a set of orthogonal Latin squares, two parameters take their

values from the row and column indices as before. The remaining parameters take their values from the ordered $(k-2)$-tuple formed from the superimposed squares.

There are severe restrictions on the existence of a set of orthogonal Latin squares. It is always possible to construct a single $n \times n$ Latin square for any size n. However, as shown in [Hal86]:

1. For any n, there are at most $n-1$ mutually orthogonal $n \times n$ Latin squares.

2. If p is the smallest prime in the prime factorization of n, and occurs j times in that factorization, then there exist at least $p^j - 1$ orthogonal Latin squares.

This guarantees the existence of the maximum of $n-1$ orthogonal Latin squares when $n = p^m$ for some prime number p and integer $m \geq 1$. The cases with the maximum number of orthogonal Latin squares are of interest, because they can handle the greatest number of parameters, and because there is a useful construction algorithm for them.

Unfortunately, this result also points out a particularly problematic case whenever $n = 4m + 2$, for some integer $m \geq 1$. Since the prime factorization of n has only a single 2, the prime factorization result guarantees only the existence of a single Latin square. When $m > 1$, it has been shown that at least two orthogonal Latin squares exist, but a pair of 6×6 orthogonal Latin squares cannot be found (see [Hal86]). There are many open problems associated with the existence of orthogonal Latin squares.

In a system with k parameters and n possible values for each parameter (that is, n^k system configurations), then the method of orthogonal Latin squares will reduce the number of system configurations to $\max(n^2, (k-1)^2)$. At least $k-2$ squares are needed to handle k parameters using orthogonal Latin squares. This requires using squares of order $k-1$ to ensure the existence of a sufficient number of squares.

Figure 4-3 shows a pair of 3×3 orthogonal Latin squares:

$$\begin{array}{ccc} 0 & 1 & 2 \\ 1 & 2 & 0 \\ 2 & 0 & 1 \end{array} \qquad \begin{array}{ccc} 0 & 1 & 2 \\ 2 & 0 & 1 \\ 1 & 2 & 0 \end{array}$$

Figure 4-3: Two 3×3 orthogonal Latin squares

Figure 4-4 shows a pair of superimposed 3×3 orthogonal Latin squares, created from Figure 4-3:

$$\begin{array}{ccc} (0,0) & (1,1) & (2,2) \\ (1,2) & (2,0) & (0,1) \\ (2,1) & (0,2) & (1,0) \end{array}$$

Figure 4-4: Superimposed orthogonal Latin squares

Since there is no duplicate ordered pairs in Figure 4-4, each of the nine possible ordered pairs is covered. Furthermore, they are covered exactly once each. Therefore, the squares in Figure 4-3 are a set of orthogonal Latin squares.

4.4.3 Algorithm for construction of orthogonal Latin squares

A construction algorithm for a set of orthogonal Latin squares is presented in [Gi76], and adapted here. This algorithm uses results from the theory of finite fields in algebra, and is originally due to Bose [Bos38].

Definition 4-4 Galois (finite) field

> Let p be a prime number, and let m be an integer such that $m \geq 1$. A finite field with p^m elements is called a Galois field of order p^m, and is denoted by $GF(p^m)$. □

For any prime p, and any integer $m \geq 1$, a Galois field $GF(p^m)$ exists. In the case where $m = 1$, the integers modulo p, denoted Z_p, is a Galois field of order p. For $m > 1$, the Galois field $GF(p^m)$ can be constructed from Z_p by extending the field in a manner similar to the method by which the field of real members is extended to complex numbers.

In the field of real numbers, the polynomial $q(x) = x^2 + 1$ is irreducible. However, by defining a new value i as a root of $x^2 + 1 = 0$, and using polynomials of i as field elements, the real numbers are extended to complex numbers.

Algorithm 4-1 Extension of a field

Let $Z_p[x]$ denote the set of polynomials of x with coefficients in Z_p. To extend Z_p to $GF(p^m)$, find a polynomial $q(x)$ of degree m, irreducible in $Z_p[x]$, and define α such that $q(\alpha) = 0$. Then the polynomials $Z_p[\alpha]$ with degree less than m can be used as new field elements to create $GF(p^m)$. □

Algorithm 4-2 Generate a set of orthogonal Latin squares

Let $GF(n) = \{x_0, x_1, ..., x_{n-1}\}$ be a finite field of order $n = p^m$ where p is a prime number, m is an integer ≥ 1, $x_0 = 0$ and $x_1 = 1$. Define the elements of squares $\overline{L^l} = [a^l_{uw}]$ for $1 \leq l \leq n-1$, by $a^l_{uw} = x_l \cdot x_u + x_w$, in the Galois field arithmetic, for $0 \leq u, w < n$. □

Note that this will give us a set of squares with entries in the Galois field.

Theorem 4-1 Algorithm 4-2 generates a set of $n-1$ orthogonal Latin squares

Let $\overline{L^l}$ be defined as in Algorithm 4-2, for $l \geq 1$. Then $\overline{L^l}$ is a Latin square, and $\{\overline{L^1}, \overline{L^2}, ..., \overline{L^{n-1}}\}$ are mutually orthogonal. □

Proof: (from Gilbert [Gi76])

The proof is in two parts. First, we show that each square is Latin. Second, we show that the set of squares is orthogonal.

Consider two elements in the same row of $\overline{L^l}$:

$$a^l_{uw} - a^l_{uj} = (x_l \cdot x_u + x_w) - (x_l \cdot x_u + x_j) = x_w - x_j \neq 0 \text{ if and only if } w \neq j.$$

Therefore, there are no duplicate elements in the row. Each row of size n must be a permutation of $GF(n)$.

Consider two elements in the same column:

$a^l_{uw} - a^l_{iw} = (x_l \cdot x_u + x_w) - (x_l \cdot x_i + x_w) = x_l \cdot x_u - x_l \cdot x_i = x_l\,(x_u - x_i) \neq 0$ whenever $u \neq i$, since $l \geq 1$ and therefore, $x_l \neq 0$.

Therefore, there are no duplicate pairs of elements in the column. Each column of size n must then be a permutation of $GF(n)$.

Therefore, $\overline{L^l}$ is a Latin square.

We now show that the set of squares is orthogonal

Suppose $(a^k_{uw}, a^l_{uw}) = (a^k_{ij}, a^l_{ij})$ – that is, the (u, w) element of superimposing $\overline{L^k}$ and $\overline{L^l}$ is the same as the (i, j) element.

Then $a^k_{uw} = a^k_{ij}$ and $a^l_{uw} = a^l_{ij}$. Hence $x_k \cdot x_u + x_w = x_k \cdot x_i + x_j$ and $x_l \cdot x_u + x_w = x_l \cdot x_i + x_j$.

Subtracting gives $(x_k - x_l)x_u = (x_k - x_l)x_i$ or $(x_k - x_l)(x_u - x_i) = 0$.

$GF(n)$ has no zero divisors, so either $k = l$, or $u = i$. If $k = l$, then we are superimposing a square on itself. Otherwise, $u = i$, but this means the same element occurs twice in a row - contradicting the result that L^l is a Latin square.

Therefore, $\overline{L^k}$ is orthogonal to $\overline{L^l}$.

Therefore, Algorithm 4-2 generates a set of $n - 1$ orthogonal Latin squares. □

Algorithm 4-3 Convert orthogonal Latin squares from Galois field to integer elements

Let $\overline{L^l} = [a^l_{uw}]$, as defined in Algorithm 4-2. Define $L^l = [b^l_{uw}]$ such that if $a^l_{uw} = x_q \in GF(n)$ then $b^l_{uw} = q$. The result is that L^l has entries that are in the set $\{0, \ldots, n - 1\}$. □

We are replacing a Galois field element by its index in the set of field elements, to bring the value into the range $\{0, \ldots, n-1\}$. The mapping of Galois field elements to their set indices forms a one to one correspondence.

One reason to change at this point from squares containing Galois field elements to square containing integers is that we shall no longer need the properties of the Galois field arithmetic. The conversion allows us to use standard integer arithmetic from this point. In addition, the squares are more "readable".

In the next section, we shall present an example of constructing a set of orthogonal Latin squares from a Galois field, which in turn was constructed by extending a field from modular integers.

4.4.4 Example: Construct three orthogonal Latin squares of order four

To demonstrate the use of Algorithms 4-1 through 4-3, consider the case of constructing three orthogonal 4×4 Latin squares. First, we need to generate the Galois field $GF(4) = \{x_0, x_1, x_2, x_3\}$ with $x_0 = 0$ and $x_1 = 1$.

$GF(4) = GF(2^2)$, so we start with integers mod 2, $\mathbf{Z}_2 = \{0, 1\}$. Next, we need to find a polynomial $q(x)$ in $\mathbf{Z}_2[x]$ such that $q(x)$ is of degree 2, and irreducible in $\mathbf{Z}_2[x]$. A choice for $q(x)$ is $x^2 + x + 1$, since $q(0) = 1$ and $q(1) = 1$. We define a new field element α such that $\alpha^2 + \alpha + 1 = 0$ in $GF(4)$. Then $GF(4) = \mathbf{Z}_2[\alpha] = \{0, 1, \alpha, \alpha+1\}$.

Since $x_1 = 1$, $a^1_{ij} = x_i + x_j$, and we can form Latin square $\overline{L^1}$ from the addition table for $GF(4)$:

0	1	α	$\alpha+1$
1	0	$\alpha+1$	α
α	$\alpha+1$	0	1
$\alpha+1$	α	1	0

$\overline{L^2}$ can be obtained by $a_{ij}^2 = \alpha \cdot x_i + x_j$. Note that since $\alpha^2 + \alpha + 1 = 0$ in $GF(4)$, $\alpha^2 = \alpha + 1$.

0	1	α	$\alpha+1$
α	$\alpha+1$	0	1
$\alpha+1$	α	1	0
1	0	$\alpha+1$	α

$\overline{L^3}$ can be obtained by $a_{ij}^3 = (\alpha+1) \cdot x_i + x_j$.

0	1	α	$\alpha+1$
$\alpha+1$	α	1	0
1	0	$\alpha+1$	α
α	$\alpha+1$	0	1

While the field elements in the squares are $\{0, 1, \alpha, \alpha+1\}$ we can renumber these elements as $\{0, 1, 2, 3\}$ using Algorithm 4-3. If we superimpose the three orthogonal squares with the renumbered elements, we obtain Figure 4-5 as a result:

(0, 0, 0)	(1, 1, 1)	(2, 2, 2)	(3, 3, 3)
(1, 2, 3)	(0, 3, 2)	(3, 0, 1)	(2, 1, 0)
(2, 3, 1)	(3, 2, 0)	(0, 1, 3)	(1, 0, 2)
(3, 1, 2)	(2, 0, 3)	(1, 3, 0)	(0, 2, 1)

Figure 4-5: Three superimposed 4 × 4 orthogonal Latin squares

Note that all sixteen ordered triples, and the sets of ordered pairs contained within, are distinct, so the squares are indeed orthogonal.

4.4.5 Orthogonal array construction

With the set of orthogonal Latin squares we have generated, we can now proceed to determining the set of test configurations (experiments) represented by an orthogonal array.

Each ordered triple in Figure 4-5 represents a test configuration, where parameter 0 takes the row index, parameter 1 takes the column index, and parameters 2 through 4 take the value of the ordered triple in the superimposed squares. For example, the highlighted triple in Figure 4-5 represents the test configuration $\{2_0, 0_1, 2_2, 3_3, 1_4\}$: row 2, column 0, and entry (2, 3, 1).

This process can be generalized in an algorithm to convert from a set of $n-1$, $n \times n$ orthogonal Latin squares to the orthogonal array $O(n^2, n+1, n, 2)$:

Algorithm 4-4 Generate orthogonal array $O(n^2, n+1, n, 2)$

Let n be a prime or prime power. To find the element of the array $O_{ij}(n^2, n+1, n, 2)$ for $i = 0, \ldots, n^2 - 1$; $j = 0, \ldots, n$, calculate $u = \lfloor i/n \rfloor$ and $w = i \bmod n$. The value u represents the row index in an orthogonal Latin square (with entries renumbered to integers by Algorithm 3), and the value w represents the column index. Column $j = 0$ of the orthogonal array will be the row indices for the set of orthogonal Latin squares. Column $j = 1$ of the orthogonal array will be the column indices for the set of orthogonal Latin squares. For column $j > 1$, the entry in the orthogonal array will be the entry in square $j - 1$, at row u, column w.

Therefore: $$O_{ij}(n^2, n+1, n, 2) = \begin{cases} u, \text{ for } j = 0 \\ w, \text{ for } j = 1 \\ L_{uw}^{j-1}, \text{ for } j > 1 \end{cases} \quad \square$$

Theorem 4-2 Algorithm 4-4 generates an orthogonal array

Let $O(n^2, n+1, n, 2)$ be generated by Algorithm 4-4. Then $O(n^2, n+1, n, 2)$ satisfies the properties of an orthogonal array.

Proof:

First, we examine the array size. Each row in the orthogonal array is generated as follows. Column 0 represents the row index of a position within a Latin square. Column 1 represents the column index of a position within a Latin square. The remaining columns consist of the entries in the various orthogonal Latin squares at the specified position. Since n is a prime power, a set of $n - 1$ orthogonal Latin squares can be generated by Algorithm 4-2. This accounts for $n - 1$ columns in the orthogonal array, and adding the two columns based on the square indices results in a total of $n + 1$ columns. Furthermore, each entry in a Latin square results in a row in the orthogonal array, so there are n^2 rows in the orthogonal array.

Now we will show that if one selects two columns from the array, all possible ordered pairs appear in the selected columns. To do this, we shall show that no pair of rows contains the same ordered pair. With n^2 rows in the array, this means that the rows contain n^2 distinct ordered pairs. Since there are only n^2 possible ordered pairs, it follows that each ordered pair must occur exactly once.

Select columns p, q from $O(n^2, n+1, n, 2)$ such that $0 \leq p, q \leq n$, and $p \neq q$. Without loss of generality, assume that $p < q$. There are several cases to consider:

The first case is when both p and $q > 1$. Consider the ordered pair formed by the elements in row r of the array, where a_{rp} is the element in column p and a_{rq} is the element in column q. The values u and w in L_{uw}^{j-1} are determined solely from the row index, and for fixed r, they are also fixed. Therefore, the entries in the two columns depend only on the square number of the set of orthogonal Latin squares, and represent entries in the identical row and column position in two different Latin squares. By the property of orthogonality in the set of orthogonal Latin squares, the ordered pair formed by these two elements is different from all other ordered pairs between the two selected Latin squares. Hence, the ordered pair (a_{rp}, a_{rq}) is different from that for all other rows.

The first special case is when the first two columns are selected. However, the first two columns are formed by enumerating the set of row and column indices for the set of orthogonal Latin squares. By construction, they consist of a set of n^2 distinct ordered pairs.

The remaining special case is when $p = 0$ or 1, and $q > 1$. That is, for the ordered pair (a_p, a_q), the value of a_p is a row or column index, and the value of a_q is an entry in one of the set of Latin squares.

For the case $p = 0$, let us consider rows 0 to $n - 1$ of the orthogonal array. Since the value of u represents the row index in the set of orthogonal Latin squares, the values for column 0 are constant (0), and the values for column q are row 0 of one of the set of orthogonal Latin squares. Since the square is Latin, no element is repeated twice within that row. Therefore, the ordered pairs formed by selecting column 0 and another column > 1 of the orthogonal array are the set of distinct $(0,x)$ pairs. This argument can be repeated for the next set of n rows, and so on.

For the case $p = 1$, a similar argument applies except that we consider every n^{th} row in the orthogonal array (that is, all rows where $w = (i \bmod n)$ have the same value), and use the property of a Latin square that no value is contained twice within a single column.

In summary, we have n^2 rows, each with distinct ordered pairs, and therefore all of the n^2 possible ordered pairs must be covered exactly once by the pigeonhole principle. (The pigeonhole principle is as follows: if there is a set of k items, and they can be put into k distinct equivalence classes, then there must be exactly one item in each equivalence class.) The ordered pairs are covered exactly once each, and therefore satisfy the requirement in the definition of orthogonal array (Definition 4-1) that they appear the same number of times Therefore, if $O(n^2, n+1, n, 2)$ is generated by Algorithm 4-4, $O(n^2, n+1, n, 2)$ satisfies the properties of an orthogonal array.□

Algorithm 4-4 produces orthogonal arrays that have several useful properties:

- $O_{0j}(n^2, n+1, n, 2) = 0$ for all j (i.e., row 0 is all zeros).
- $O_{ij}(n^2, n+1, n, 2) = i$, for all $i < n$ and $j > 0$ (i.e., the first n rows have entries equal to the row index, excepting column 0).
- $O_{i0}(n^2, n+1, n, 2) = \lfloor i / n \rfloor$ (i.e., column 0 consists of n zeros, n ones, etc.)
- Each column of $O(n^2, n+1, n, 2)$, except column 0, is comprised of sets of permutations of $(0, \ldots, n-1)$.

We now show that these properties always hold.

Lemma 4-1 Contents of row 0

Let $O(n^2, n+1, n, 2)$ be generated by Algorithm 4-4. Then $O_{0j}(n^2, n+1, n, 2) = 0$ for $0 \le j \le n$. That is, row 0 is all zeros.

Proof:

For $j = 0$, we have $O_{00} = \lfloor 0 / n \rfloor = 0$ from the construction for u in Algorithm 4-4.

For $j = 1$, we have $O_{01} = (0 \bmod n) = 0$ from the construction for w in Algorithm 4-4.

For $1 < j \le n$, we have $O_{0j} = L_{00}^{j-1}$, as we have just shown that $u = 0$ and $w = 0$. By the definition of $\overline{L^l}$, we are interested in the values a_{00}^l for all l. Now, $a_{uw}^l = x_l \cdot x_u + x_w$, so $a_{00}^l = x_l \cdot x_0 + x_0$. By the rules for the Galois field in Definition 4-4, $x_0 = 0$, so the value of $a_{00}^l = x_l \cdot 0 + 0 = 0$ for any l. However, if $a_{uw}^l = x_q$ then $b_{uw}^l = q$. We have $a_{00}^l = x_0 = 0$ for all l, so $b_{00}^l = 0$ for all l. Therefore, $O_{0j} = 0$ for all $j > 1$.

Thus, we have $O_{0j}(n^2, n+1, n, 2) = 0$ for $0 \le j \le n$. □

Lemma 4-2 Contents of first n rows, except for column 0

Let $O(n^2, n+1, n, 2)$ be generated by Algorithm 4-4. Then $O_{ij}(n^2, n+1, n, 2) = i$, for all $i < n$ and $0 < j \leq n$. That is, except for column 0, the first n rows consist of all zeros, all ones, and so on up to $n - 1$.

Proof:

If $i < n$, then $\lfloor i / n \rfloor = 0$. Therefore, we have $u = 0$. Furthermore, since $i < n$, $i \bmod n = i$, and therefore, $w = i$.

When $j = 1$, we have $O_{i1}(n^2, n+1, n, 2) = w$, so $O_{i1}(n^2, n+1, n, 2) = i$.

For $i < n$ and $j > 1$, we have $O_{ij} = L_{0i}^{j-1}$, as we have just shown that $u = 0$ and $w = i$. By the definition of L^l, we are interested in the values a_{0i}^l for all l. Now, $a_{0i}^l = x_l \cdot x_0 + x_i$. By the rules for the Galois field in Definition 4-4, $x_0 = 0$, so the value of $a_{0i}^l = x_l \cdot 0 + x_i = x_i$ for any l. Therefore, $b_{0i}^l = i$. Therefore, $O_{ij} = i$ for all $j > 1$.

Thus, we have $O_{1j}(n^2, n+1, n, 2) = i$ for all $i < n$ and $j > 0$. □

Lemma 4-3 Contents of column 0

Let $O(n^2, n+1, n, 2)$ be generated by Algorithm 4-4. Then $O_{i0}(n^2, n+1, n, 2) = \lfloor i / n \rfloor$. That is, column 0 consists of n zeros, n ones, and so on.

Proof:

$O_{i0}(n^2, n+1, n, 2) = \lfloor i / n \rfloor$ is true by construction, as this is the value of u. □

Lemma 4-4 Contents of each column, except column 0

Let $O(n^2, n+1, n, 2)$ be generated by Algorithm 4-4. Then each column of $O(n^2, n+1, n, 2)$, except column 0, is comprised of sets of permutations of $(0, \ldots, n - 1)$.

Proof:

Consider $O_{ij}(n^2, n+1, n, 2)$ for all $j > 0$.

When $j = 1$, we have $O_{i1}(n^2, n+1, n, 2) = w = i \bmod n$. Therefore, column 1 will be the values 0, ..., $n - 1$; 0, ..., $n - 1$, repeated for n^2 rows.

When $j > 1$, we have $a_{uw}^{j-1} = x_{j-1} \cdot x_u + x_w$. The values of u and w are independent of j; they depend only on i. We have $u = \lfloor i / n \rfloor$ and $w = i \bmod n$. Because of the floor function in the construction of u, the effect is to have integer division by n. Therefore, as i increases, the first n values of u will be the same (= 0), the next n values of u will be the same, and so on. In the construction of w, the modulus function has the effect of taking the remainder when dividing by n. Therefore, the value of w will proceed from 0 to $n - 1$, as i increases, and then go back to 0. Now, let us look at $a_{uw}^{j-1} = x_{j-1} \cdot x_u + x_w$. For i taking the values κn through $(\kappa + 1)n - 1$ for some integer $\kappa \geq 0$, we have $x_{j-1} \cdot x_k + x_i$. By the properties of an algebraic field, if $ab + c = ab + d$, then $c = d$. The product $x_{j-1} \cdot x_k$ remains fixed as i varies, and for the values κn through $(\kappa + 1)n - 1$, x_i will take on each of the Galois field elements $\{x_0, x_1, \ldots, x_{n-1}\}$. Because a Galois field is closed under addition and multiplication, and there are a finite number of elements, the value of $x_{j-1} \cdot x_k + x_i$ will also take on each of the Galois field elements $\{x_0, x_1, \ldots, x_{n-1}\}$. Since b_{uw}^{j-1} is constructed by taking the index of the Galois field elements, the values of b_{uw}^{j-1} will take on each of the values from the set $\{0, \ldots, n - 1\}$. Therefore, each collection of i consecutive column entries forms a permutations of $(0, \ldots, n - 1)$.

Thus, for fixed $j > 0$, the value of $O_{ij}(n^2, n+1, n, 2)$ will consist of a set of n permutations of $(0, \ldots, n - 1)$. □

Figure 2-1 shows the orthogonal array $O(16, 5, 4, 2)$ that is constructed from the set of three 4×4 orthogonal Latin squares shown in Figure 4-5:

0	0	0	0	0
0	1	1	1	1
0	2	2	2	2
0	3	3	3	3
1	0	1	2	3
1	1	0	3	2
1	2	3	0	1
1	3	2	1	0
2	0	2	3	1
2	1	3	2	0
2	2	0	1	3
2	3	1	0	2
3	0	3	1	2
3	1	2	0	3
3	2	1	3	0
3	3	0	2	1

Figure 4-6: The orthogonal array $O(16,5,4,2)$

4.5 Relevant previous work related to orthogonal arrays

An overall description of engineering quality principles using orthogonal arrays to set up experiments is contained in [Ro96]. The primary advocate of using the techniques of experimental design for quality control has been Taguchi. Taguchi has written a two-volume work [Ta87] concentrating on experimental design, with the emphasis on orthogonal arrays for setting up experiments.

The following are well-known constructions of orthogonal arrays $O(\rho, k, n, d)$, from the literature of combinatorics:

- $O(n^2, n+1, n, 2)$ Bose (1938) [Bos38].

  This is the basic construction from a set of orthogonal Latin squares, from which Algorithm 4 has been adapted.

An additional requirement is that n is a prime power. That is, $n = p^m$ for some prime p and some integer $m \geq 1$. The upper limit on the number of columns is $n + 1$, but an arbitrary number of columns can be removed if there are fewer parameters.

- $O(n^d, n + 1, n, d)$ Bush (1952) [Bus52]

  This construction allows for orthogonal arrays of arbitrary strength d, which cover all d-way interactions. The number of configurations increases from n^2 to n^d.

- $O(\lambda n^2, \lambda n + 1, n, 2)$ Bose, Bush (1952) [Bos52]

  This construction provides for an orthogonal array that handles an arbitrary number of parameters with a suitable choice of λ.

- $O(2n^3, 2n^2 + 2n + 1, n, 2)$ Addelman and Kempthorne (1961) [Ad61]

Algorithms for the above have been implemented in the C programming language by Owen [Ow]. Another available generator of orthogonal arrays is for the SAS statistical package, implemented by Tobias [To].

Orthogonal arrays for arbitrary values of ρ, k, n, and d do not necessarily exist. As with many combinatorial results, particular results are on a case-by-case basis. There are only a few general results for the existence, or non-existence of orthogonal arrays. An important paper that defines several bounds among ρ, k, n, and d, including the ones in section 4.4.2 is by Rao (1947) [Ra47].

4.6 Chapter summary

In this chapter, we have looked at statistical experimental design in general. We have reviewed the standard orthogonal array used in the construction of a set of experiments. We have presented an algorithm for constructing orthogonal arrays, and shown an example of the use of the algorithm. We have also reviewed previous work related to orthogonal arrays.

In the next chapter, we look at the specific properties of the context of software testing, and see how these properties affect the construction of a set of test configurations.

Chapter 5 Applying experimental design to software system testing

5.1 Chapter introduction

In Chapter 4, we have looked at statistical experimental design in general, including the use of orthogonal arrays to generate a set of test configurations. However, we did not look at the specific problem of software system component testing to see if this problem has any specific properties that allow for optimization. We shall answer this question in this chapter.

This chapter is organized as follows. First, we discuss the principles of experimental design, and the properties of system testing. The goals are to identify common elements, and highlight the differences from general experimental designs. Second, we define covering arrays, which are constructs similar to orthogonal arrays. However, a covering array is more suited to generate test configurations, because the covering array more closely captures the interaction test coverage criterion defined in section 3.2.3. Finally, we review previous work on the theory of covering arrays and their application to testing.

5.2 Applying the principles of experimental design to software system testing

We can directly make use of the principles of experimental design to determine system test configurations. The software testing "experiment" that is proposed is to determine if all of the software components (factors) are truly free of unwanted interactions. A successful run of the experiments would confirm this independence.

One possible goal is to exercise interactions caused by (up to) a specific number of parameters. In particular, covering of two-way interactions seems to be a reasonable goal that balances the requirement of having a practical number of test configurations, while still finding most of the potential interaction problems in a system. Results obtained in an empirical study [Coh96] show that excellent code coverage can be achieved with two-way interactions. If required, experiments can be designed to test interactions among

three or more specific parameter values, at the expense of an increased number of configurations. However, to generate a set of configurations that cover all d-way interactions, the number of configurations will be proportional to at least n^d.

The application to software system testing turns out to be simpler than the general situation of experimental design, and this allows some of the constraints to be removed. This means that smaller designs with fewer configurations can be used while still achieving the goals of software testing.

An orthogonal array has a "balance" property. This means that each parameter value appears in the same number of test configurations. Furthermore, each d-way interaction must also appear in an identical number of configurations. This facilitates the analysis of results from a range of continuous, real-valued numbers. (Typically, such results would be obtained by measurements during an experiment.)

Two simplifying considerations arise in applying statistical experimental design to software testing. The first is that the values for each system configuration parameter are typically selected from a set of discrete elements. For example, a parameter value selection could be which model of telephone to use for placing a phone call. The sets of factors and levels are readily apparent, and there is no need to select a set of values from a continuous real-valued domain, as would be the case with traditional experimental design.

Secondly, a statistical analysis of real-valued results is not required. It is assumed that a "reasonably complete" test suite is available for each potential configuration. The expected test results are determined directly (e.g., from specifications or customer requirements.) After running each test case, a test verdict is assigned as one of "pass," "fail," or perhaps "inconclusive" if permitted by the test execution environment (for example, an inconclusive verdict could be assigned if a test preamble fails). An experiment corresponds to running the entire test suite for one configuration of system parameters, so an overall verdict for the entire test suite is needed to represent the result of the experiment. The assumption is then that if pass verdicts result for all experiments –

that is, all executions of test suites for all configurations in the design – the probability of unwanted interactions is small (depending on the coverage of the test suite used for each configuration). A discrete result for each experiment can be determined on an individual basis. There is no need to analyze a set of continuous, real-valued results, with confidence intervals and so on.

Assuming we have prior unit testing, and discrete test verdicts, the requirement for balance can be removed. If every possible d-way combination of values is covered at least once, any undesirable interactions between those values could be detected (with a suitable test suite.) Removing the need for balance provides for a significant reduction in the number of test configurations required.

The orthogonal array turns out, in a sense, to be not quite appropriate for software system testing. In one sense, it is too weak, while in another sense it is too strong.

The weakness of the orthogonal array is due to the limitations that are associated with their existence. Only arrays of certain shapes exist. Furthermore, they normally are based on the assumption that all parameters must have the same number of values. For practical use, these limitations must be addressed, and we investigate them in Chapter 8.

The orthogonal array is too strong because it enforces the condition of balance referred to earlier. That is, all parameter values are required to appear in an identical number of configurations.

However, if we restrict our goal to just coverage of pair-wise (or d-way) interactions, we can drop the requirement for balance, and therefore reduce the number of configurations. To do this, we will use a slightly different construction.

The construct providing pair-wise coverage, without requiring balance, is called a *covering array* [St98b].

5.3 Definition of covering array

Definition 5-1 Covering array

A *covering array* is denoted by $C(\rho, k, n, d)$, where:

- ρ is the number of rows in the array. The k-tuple forming each row represents a single test configuration, and thus ρ represents the number of test configurations.
- k is the number of columns, which represents the number of parameters.
- The entries in the array are the values 0, ..., $n-1$, where $n = f(n_0, \ldots, n_{k-1})$. Typically, this means that each parameter would have (up to) n values.
- d is the strength of the array (see below).

A covering array has *strength d* if in any $\rho \times d$ sub-matrix (that is, selecting d columns), each of the n^d possible d-tuples (rows) appears at least once. □

This is similar to the definition of an orthogonal array (see Definition 4-1), where each d-interaction element appears not only at least once, but also the same number of times. Thus, an orthogonal array satisfies the requirements of a covering array, but a covering array may not be orthogonal.

However, the covering array does preserve the properties identified in section 4.3.1. That is, we can arbitrarily interchange the rows or columns of a covering array. We can also renumber the values. Most importantly, we can also delete columns that are not needed.

The definition of strength captures the desired interaction element coverage property, reproduced from Definition 3-17 below. (Recall that S is a set of selected test configurations, and T is the set of all possible configurations.)

The goal is to find $S \subseteq T$, such that every element in X_d belongs to at least one member of S.

The correspondence is as follows:

- The ρ rows of the covering array correspond to S, the set of selected test configurations.
- Selecting d columns for a single row in the covering array corresponds to constructing an index set σ_d, and then determining the corresponding interaction elements $\chi(\sigma_d, \tau)$.
- The set X_d of all interaction elements corresponds to each of the possible d-tuples that can be formed from the values for every possible selection of d columns.
- Since each of the possible d-tuples for any selection of d columns must occur in one of the ρ rows, this corresponds to requiring that every member of X_d be covered by at least one element of S'.

With a covering array, the capability to isolate and extract the effect of a single parameter is lost for analysis of real-valued results because the balance property is no longer present. This capability is not necessary when discrete test verdicts are derived independently. However, testing at the system level normally occurs in practice after each software component has been tested as an individual unit. If each software component has passed unit testing, it is not necessary to extract the effect of a single parameter.

Therefore, the covering array preserves all of the properties needed for software testing, and we can reduce the number of configurations required, compared to an orthogonal array.

We shall be especially interested in covering arrays of strength two: in such an array, one can select any two columns, and every possible 2-interaction element occurs at least once in the rows formed by those two columns.

As with orthogonal arrays, covering arrays do not exist for arbitrary values of ρ, k, n, and d. Our interest is finding covering arrays where the value of ρ is as small as possible, for chosen values of k, n, and d. This goal means that we are trying to cover all d-way interactions in as few configurations as possible.

This nomenclature reflects the idea that in an orthogonal array, one should be able to detect the influence of any parameter independently of any other parameter. Using a covering array for software testing, we need only cover the set of interactions among the parameters without regard to having to analyze values that are real numbers. The result analysis is provided instead by the discrete-valued verdict ("pass," "fail," or possibly "inconclusive") produced after executing a test suite.

With a covering array, the correspondence between a combinatorial design (such as orthogonal Latin squares) and the equivalent orthogonal array is lost. That is, there is no equivalent combinatorial design to a covering array.

The existence of a set of orthogonal Latin squares will generate an orthogonal array, and thus a covering array. However, the reverse is not true: there are covering arrays that have no corresponding set of orthogonal Latin squares (or any other design). An example of such an array (adapted from [St98b]) is shown in Figure 5-1:

$$\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{array}$$

Figure 5-1: The covering array $C(5, 4, 2, 2)$

In this example, it is apparent that the entries need not be balanced, as there are more 1's than 0's in the array. However, if you select any two columns, you will find that each ordered pair (0, 0), (0, 1), (1, 0), and (1, 1) is covered at least once. Therefore, this array meets our software testing coverage goals. If an experiment constructed according to this array produced real-valued results, it would be difficult to analyze because the factor levels are unbalanced. When testing software, we can resort to checking the results against the software specification and do not have to use statistical methods. Thus, the lack of balance is not an impediment.

Using a covering array can reduce the number of test configurations required, because it is less restrictive. We have already noted that a pair of 6×6 orthogonal Latin squares does not exist, and therefore $C(36, 4, 6, 2)$ cannot exist either. Suppose one has four parameters, each with six possible values. This is the situation where one would need $C(36, 4, 6, 2)$. To construct an experiment based on an orthogonal array, one could instead use a pair of 7×7 orthogonal Latin squares to construct $O(49, 8, 7, 2)$ which has 49 test configurations. Two adjustments are then needed to fit to the problem. Four of the parameter columns would be dropped, to reduce the array to four parameters. The "seventh value" for each parameter could be either a "don't care" value, or an extra level for each factor.

However, we can construct $C(37, 4, 6, 2)$, a covering array for four parameters with six values, and it has only 37 configurations [St98b]. The array can be constructed by using two "incomplete" orthogonal Latin squares. Two 6×6 squares, with the upper left 2×2 grid removed, are constructed so that the remaining "squares" have similar "orthogonal" and "Latin" properties. There is, at most, a single instance of each value in any row and column of the squares. The 32 ordered pairs formed by superimposing the remnants of the squares contain no duplicate ordered pairs.

Furthermore, the "hole" has the property that only two values are "missing," and that the hole is in the position where the row and column indices match the missing values. Figure 5-2 illustrates these squares.

```
* * 2 4 5 3     * * 2 5 3 4
* * 5 3 2 4     * * 4 3 5 2
2 3 4 5 0 1     4 5 0 1 2 3
3 2 1 0 4 5     2 3 5 4 1 0
4 5 3 2 1 0     3 2 1 0 4 5
5 4 0 1 3 2     5 4 3 2 0 1
```

Figure 5-2: Two incomplete 6×6 orthogonal Latin squares

Except for the entries indicated with a *, these are two orthogonal Latin squares. Furthermore, the entries that are missed in the holes are the values 0 and 1 in both

squares. Finally, the hole is also in the position where the row and column indices are 0 and 1. It is not possible to insert the 0 and 1 entries in a way where the squares are both Latin (a single 0 and 1 per row and column) and orthogonal (all four ordered pairs are created by superimposing the squares.)

However, by generating an array from the 32 entries outside the holes, and appending the covering array $C(5, 4, 2, 2)$ from Figure 5-1, we can create the complete covering array $C(37,4,6,2)$, as shown in Figure 5-3. The 37 configurations in this array compare with the 49 configurations required by the orthogonal array.

0	2	2	2		4	0	4	3
0	3	4	5		4	1	5	2
0	4	5	3		4	2	3	1
0	5	3	4		4	3	2	0
1	2	5	4		4	4	1	4
1	3	3	3		4	5	0	5
1	4	2	5		5	0	5	5
1	5	4	2		5	1	4	4
2	0	2	4		5	2	0	3
2	1	3	5		5	3	1	2
2	2	4	0		5	4	3	0
2	3	5	1		5	5	2	1
2	4	0	2		0	0	0	0
2	5	1	3		1	1	1	0
3	0	3	2		1	1	0	1
3	1	2	3		1	0	1	1
3	2	1	5		0	1	1	1
3	3	0	4					
3	4	4	1					
3	5	5	0					

Figure 5-3: The covering array $C(37, 4, 6, 2)$

The study of incomplete mutually orthogonal Latin squares ("MOLS with holes") is first addressed in [DS83], and the description here is adapted from [He97]. At this point, there are some general approaches for constructing incomplete squares, but as with much of combinatorics, specific results are on a case-by-case basis.

The use of a covering array in place of an orthogonal array is particularly useful in the case where there are many parameters and few values. The set of orthogonal Latin squares is restricted to cardinality of one less than the number of values. When the number of parameters exceeds the number of values by at least one, the use of a covering array can significantly reduced the number of required configurations.

5.4 Previous work

This section reviews previous work on covering arrays. The first subsection looks at work that relates the theory of covering arrays to testing. The second subsection discusses related applications, tools, and algorithms.

5.4.1 Previous work on covering array theory applied to testing

As most of the work in statistical experimental design has required the balance property, the literature on covering arrays is not extensive.

Boroday and Grunskii (1992) [Bor92] identify two key properties of covering arrays. First, it is an early indication of the potential for using covering arrays for testing. Second, they prove several transformations of covering arrays (such as interchanging rows or columns) that preserve the property of being a covering array.

Williams and Probert (1996) [Wi96] provide a guide to the calculation of a set of test configurations that achieve pair-wise coverage using a deterministic approach. The paper contains the preliminary results of the work described here. Updated results are provided in Williams (2000) [Wi00] and in Williams and Probert (2001) [Wi01].

Stevens and Mendelsohn (1998) [St98a] [St98b] provide a theory for the recursive construction of covering arrays. Unlike many papers in the combinatorial literature, this

paper focuses on covering arrays, rather than the combinatorial designs used typically used to generate them. Several construction approaches are introduced. The paper [St98b] contains a table of the smallest known covering arrays (found by a computer search using a simulated annealing algorithm). This is discussed further in section 9.3.4.

The Stevens and Mendelsohn paper focuses on the case where there are three or more values for each parameter, because the case where every parameter has two values has been solved completely. A paper by Sloane (1993) [Sl93] on the application of covering arrays to intersecting codes, gives a construction (although not an algorithm) for the two-value case based on eigenvectors of the set of all $\left\lceil \frac{k}{2} \right\rceil$-subsets of a $(k-1)$-set. The result is that for k parameters, each having two values, the minimum size of a covering array is $\binom{k-1}{\left\lceil \frac{k}{2} \right\rceil}$. This result was obtained independently by Kleitman and Spencer [Kl73], and Katona [Kat75].

However, a general result for the minimum size of covering arrays for three or more parameters is an open problem. The existence of various combinatorial designs to generate covering arrays has largely been determined on a case-by-case basis. As previously noted, the existence of a design is sufficient but not necessary for the existence of a covering array, and a general minimal covering array size remains an open problem. Some lower bounds have been determined by Sloane [Sl97], and Stevens, Moura, and Mendelsohn [St98c], but constructions that meet these bounds have not been achieved in many cases.

Another implementation of a generator of covering arrays is by Hardin and Sloane (1993) [Har93]. Sloane's interest is in generating error-correcting codes, although the tool can be used for any generation of experimental designs. The program is called "gosset," [Har92] and it uses Monte Carlo searching techniques.

Literature on covering arrays of degree higher than two is scarce. An approach for covering arrays of degree 3 has been recently been identified by Chateauneuf, Coulbourn, and Kreher [Cha99].

5.4.2 Previous work on covering array tools

[Note: Some of the papers in this section were also discussed in section 2.6, in cases where the paper contains discussions of both tools and applications. The commentary in this section is related to the tools, and not on specific application of those tools.]

A number of experience reports are available on the use of Telcordia's AETG (Automatic Efficient Test Generator) system, developed by Cohen, Dalal, Parelius, and Patton. AETG is described in [Coh94a], [Coh94b], [Coh96], and [Coh97]. The second report, [Coh94b] focuses on application of the method only, and was discussed in section 2.6.

The first report (1994) [Coh94a] introduces the idea of a combinatorial approach to test generation based on statistical experimental design techniques. The use of the system is then described in that the user has to enter the set of fields (parameter values) and relations (dependencies among parameter values). It is noted that AETG can also generate test configurations for *d*-way interactions, and not just pair-wise interactions. The significance of this paper is that it is the first recognition that a full orthogonal array is not necessary for software testing purposes, and that significantly fewer configurations can be used if one relaxes the requirement of balance. They also point out that suitable orthogonal arrays do not typically exist for an arbitrary problem. As for how AETG determines its designs, the algorithms are declared as Telcordia proprietary. However, there are two clues that heuristics may be involved. The first is that one of the designs presented is for 13 parameters with three values each. There are 19 configurations, and there does not appear to be a coherent pattern to the array – which does not mean that one does not exist, of course! The second clue is that there is a capability of adding to a set of pre-existing tests to bring it up to covering *d*-way interactions.

This report is updated in [Coh97] (1997). It appears that the AETG tool has been improved, as the same case study is presented, but with fewer configurations. This paper gives some information about the algorithms that are used. The test configurations are generated using a greedy heuristic, where the objective is to find the maximum number of uncovered pairs at any stage. There is also a random element to the heuristic, and therefore exact results are not predictable in advance.

A summary of the algorithm to add a test case follows:

Algorithm 5-1: AETG's algorithm to add a new configuration

1. Choose a parameter, and a value for that parameter, such that the parameter value appears in the greatest number of uncovered pairs.

2. Choose a random order for the remaining parameters.

3. For each parameter do:

   For each possible value for the new parameter, chose the value that covers the most pairs with the previously assigned parameters. □

The tool produces a set (the size 50 is mentioned in the paper) of candidate test cases at each stage, and then chooses the best test case from among the candidate set. The results available in the candidate set vary because of the random order in which the parameters are considered in step 3. It would appear that this approach would be rather time-consuming.

In the third report by the creators of AETG (1996) [Coh96], the emphasis is threefold. First, there is a definition of pair-wise coverage. Second, there is a description of how to identify parameters and values in various types of testing. Third, test coverage results are provided. This latter section contains the most interesting results. The pair-wise coverage approach was used to test the code for 10 UNIX commands. Tests produced by the AETG tool were run, and control and data flow code coverage was measured during test execution. For the command with the largest number of lines of code (the UNIX "sort" command), here are the average coverage results that were obtained:

- Block coverage: 93.5%
- Decision coverage: 83%
- Computational uses: 76%
- Predicate uses: 73.5%

These are extremely favourable results, and provide evidence that covering two-way interactions is a goal that is sufficient to provide excellent coverage. Coverage of interactions for $d > 2$ may not be worth the extra number of tests required, as compared with the additional coverage that is gained.

AETG experience is reported from a group outside the AETG development team in Burroughs, et al. (1994) [Bur94]. The method is used to set up system configurations for primary rate ISDN. In the first application to test call rejection, there were five parameters, with 7, 6, 3, 2, and 2 values each. In the second case of testing channel negotiation, there were four parameters with 3, 6, 3, and 2 values each. The AETG system was used to generate the configurations to achieve pair-wise coverage. The numbers of configurations for the two applications were 42 and 18, respectively. Achieving pair-wise coverage between the parameters with the two highest numbers of values requires at least the product of those numbers of values. It is noted that, in comparison with the use of an orthogonal array, the element of balance is sacrificed in favour of fewer configurations. Because the configurations produced by AETG had "several cases of major imbalance," it was decided to improve the balance manually ("tedious and time consuming"), while retaining the coverage goal. This is an indication that some testers are uncomfortable with the idea of unbalanced configurations. In fact, as noted in [Coh96] there can be advantages to unbalanced configurations, if inputs for specific fields are more important or more expensive than other inputs. The fears expressed in the paper regarding the lack of balance can be discounted, in light of the other papers noted above for which this was not a problem.

Another AETG experience report is provided by Dunietz, et al. (1997) [Dun97]. The paper reports on an experiment where code coverage measurements are taken for d-way parameter interaction interactions, for $d = 1$ up to k, the number of parameters. The result is that at degree 2 coverage that number of blocks executed rose to the equivalent number of blocks as the number executed by exhaustive partition testing. When the measure was changed to the number of unique paths executed, equivalence with exhaustive path testing was reached at degree four coverage. However, the use of experimental design techniques was shown to be clearly superior to random configuration sets of

configurations of equivalent size. For example, in one situation, a set of 10 configurations was sufficient to cover all pair-wise interactions. Randomly chosen sets of configurations of size 10 tended to have degree 2 interaction coverage of 76%, and the median number of blocks and paths covered was smaller.

A heuristic approach to generating covering arrays is presented by Sherwood (1994) [Sh94] in the description of the CATS (Constrained Array Test System). This tool uses a greedy search algorithm that looks only one step ahead in attempting to fill the covering array. It does not take a global, long-term approach.

Another use of a greedy heuristic is provided by Ozmirzak [Oz94] for verifying simulation parameters. In this case, pair-wise interactions between simulation (real-valued) parameters are used as the goal to verify simulations of real-time systems. The approach in this work is to formulate the interaction test coverage problem as a series of constraints, which is similar to the approach shown in section 3.4.2. A dynamic programming approach is used to solve the constraint system. The problem discussed here is different because we are not using real-valued parameters, and thus the problem becomes an integer programming problem. We investigate the quality of a linear programming approximation (i.e., one with real-valued variables) in sections 7.4 and 7.5.

Another technique for constructing covering arrays of strength 2 is by Lei and Tai [Le98]. They use a strategy called "in-parameter order" (IPO) for generating pair-wise test sets. (This approach will be discussed in greater detail in section 7.2.) The approach uses separate heuristic algorithms for vertical (adding configurations) and horizontal (adding more parameters) growth. The vertical growth algorithm is shown to be locally optimal. Two horizontal growth algorithms are presented. One algorithm is exponential in time, but is shown to be optimal in adding the fewest number of configurations required to achieve pair-wise coverage with the single new parameter. The other is polynomial in time but adds a non-optimal number of configurations. The strategy is strictly local in the sense that one parameter is added at a time, and configurations are added as necessary for each parameter. There is no overall strategy based on knowing the number of parameters in advance.

5.5 Chapter summary

In all of the previous work, there are no cases (other than our own, and that of Brett Stevens) where a recursive approach is used to generate covering arrays. Instead, a variety of heuristics has been used, with varying degrees of success. The proposed research is to investigate deterministic approaches to generating a set of test configurations. These types of algorithms will produce results faster, with at least comparable, if not fewer, numbers of test configurations. Furthermore, the number of test configurations will be predictable in advance. We especially want to ensure that as few restrictions as possible are placed on the user. Therefore, we want to have algorithms to handle any possible number of parameters or numbers of parameter values. This specifically includes cases where there are dependent parameters, differing numbers of parameter values, and so on.

This chapter has identified how the specific properties of the testing problem can be used to improve upon the results from the general technique of experimental design. Since test execution results in discrete test verdicts, and we assume that the effect of changing a single parameter is covered by other stages of testing, the constructions used to generate system test configurations need not keep the property of balance. With no need to analyze real-valued results, the construction used to generate test configurations is required only to satisfy the interaction coverage test criterion from section 3.2.3. The covering array meets this goal.

We have also looked at how covering arrays differ from orthogonal arrays. There are significantly fewer restrictions on covering arrays. As a result, more arrays are available to provide a closer fit to the problem at hand. In cases with many parameters, the covering array usually results in significantly fewer test configurations.

We have also confirmed that the covering array meets the interaction test coverage criterion, as described in section 3.2.3.

In the next chapter, we investigate the recursive construction of covering arrays of strength 2.

Chapter 6 An algorithm for covering array construction

6.1 Chapter introduction

In this chapter, we show how to construct covering arrays of strength 2 that fit a particular interaction test coverage problem. Portions of the following have been published in [Wi96] and [Wi00].

This chapter is ordered as follows. First, we show how to construct covering arrays with arbitrary width of strength 2. This is a different process from constructing orthogonal arrays, and we show how a recursive approach can produce significant reductions in the number of test configurations. Detailed algorithms are provided, as well as proofs that the algorithms generate the required covering arrays. We then discuss the number of configurations that are generated. Then, the complexity of the algorithm for construction of strength 2 covering arrays is developed.

6.2 Covering array construction

To construct a covering array of strength $d = 2$, we are given k, the number of parameters, and that parameter i has n_i values, which are enumerated as $0, \ldots, n_i - 1$. Without loss of generality, assume that the parameters are ordered so that $n_0 \geq n_1 \geq \ldots \geq n_{k-1}$.

Let n be the least integer where $n \geq n_0$ and $n = p^m$ for some prime number p and integer $m \geq 1$. This will be our definition of the function $n = f(n_0, \ldots, n_{k-1})$, as described in Definition 3-3 (section 3.2.1). We use this definition to ensure that we can find a Galois field of size n, as required by Algorithm 4-4, so that we can find an orthogonal array of strength 2 for n values.

The orthogonal array $O(n^2, n + 1, n, 2)$ is constrained to (a maximum limit of) $k = n + 1$ parameters. When $k > n + 1$, then there are two approaches that can be taken. One is to use a value of $n = k - 1$, instead of setting n as a function of the number of values. This will cause the number of configurations to grow to be order $O(k^2)$. The number of configurations is proportional to the square of the number of parameters. However, it is

possible to reduce the number of test configurations to be proportional to the logarithm base $n + 1$ of the number of parameters.

In this section, we will show how to construct covering arrays of strength 2 for any number of parameters using a recursive scheme. The full covering array is constructed from a series of smaller building blocks. In the next section, we introduce these building blocks.

6.2.1 Building blocks

Two additional types of arrays are introduced, based on the orthogonal array. These "basic" and "reduced" arrays will be used as building blocks for construction of larger covering arrays. These constructions assume that Algorithm 4-4 has been used to generate $O(n^2, n+1, n, 2)$, and that n is an integer such that $n = p^m$ for some prime number p and integer $m \geq 1$.

Definition 6-1 Basic array, $B(n^2 - 1, n + 1, n, t)$

Let $B(n^2 - 1, n + 1, n, t)$ be the array constructed by taking $O(n^2, n + 1, n, 2)$, removing the first row, and duplicating each column t times consecutively. □

The array $B(n^2 - 1, n + 1, n, t)$ has $n^2 - 1$ rows and $(n + 1) \times t$ columns. Note that a value for the strength is not included in the notation for basic arrays.

Lemma 6-1 Coverage of B array

Select columns i, j from $B(n^2 - 1, n + 1, n, 1)$ such that $0 \leq i, j \leq n$, and $i \neq j$. Then all pair-wise combinations between the columns are covered, except (0, 0).

Proof:

The basic array $B(n^2 - 1, n + 1, n, 1)$ is constructed from $O(n^2, n + 1, n, 2)$ by removing row 0. As shown in the proof of Theorem 4-2, every ordered pair must occur exactly once if two different columns are selected from $O(n^2, n + 1, n, 2)$. By Lemma 4-1, we have $O_{0j}(n^2, n + 1, n, 2) = 0$ for all j. Therefore, if we select columns i and j from $O(n^2, n + 1, n, 2)$, the ordered pair formed by the first row

must be (0, 0). If the first row is removed, then all ordered pairs except (0, 0) must be covered.

Therefore, in $B(n^2 - 1, n + 1, n, 1)$, all pair-wise combinations between any two different columns are covered, except (0, 0). □

Definition 6-2 Reduced array, $R(n^2 - n, n, n, t)$

Let $R(n^2 - n, n, n, t)$ be the array constructed by taking $O(n^2, n + 1, n, 2)$, removing the first n rows, removing column 0, and duplicating each (remaining) column t times consecutively. □

$R(n^2 - n, n, n, t)$ has $n^2 - n$ rows and $n \times t$ columns. Note that a value for the strength is not included in the notation for reduced arrays.

Lemma 6-2 Coverage of R array

Select columns i, j of $R(n^2 - n, n, n, 1)$ such that $0 \leq i, j < n$, and $i \neq j$. Then all pair-wise combinations between the columns are covered, except those of the form (x, x).

Proof:

The reduced array $R(n^2 - n, n, n, 1)$ is constructed from $O(n^2, n + 1, n, 2)$ by removing column 0 and the first n rows. Because the column 0 of $O(n^2, n + 1, n, 2)$ is removed to form $R(n^2 - n, n, n, 1)$, selecting columns i and j from $R(n^2 - n, n, n, 1)$ corresponds to selecting columns $i + 1$ and $j + 1$ from $O(n^2, n + 1, n, 2)$, where $0 < i, j \leq n$.

As shown in the proof to Theorem 4-2, every ordered pair must occur exactly once if two different columns are selected from $O(n^2, n + 1, n, 2)$. By Lemma 4-2, we have $O_{ij}(n^2, n + 1, n, 2) = i$, for all $i < n$ and $j > 0$. Therefore, if we select columns $i > 0$ and $j > 0$ from $O(n^2, n + 1, n, 2)$, the ordered pairs formed by the first n rows must be (0, 0), (1, 1), ..., $(n - 1, n - 1)$. If the first n rows are

removed, then all ordered pairs except (0, 0), (1, 1), ..., ($n-1$, $n-1$) must be covered.

Therefore, in $R(n^2-n, n, n, 1)$, all pair-wise combinations between any two different columns are covered, except (0, 0), (1, 1), ..., ($n-1$, $n-1$). □

Figure 6-1 shows an example of an orthogonal array $O(9, 4, 3, 2)$, and shows a corresponding basic array $B(8, 4, 3, 1)$ and reduced array $R(6, 3, 3, 1)$

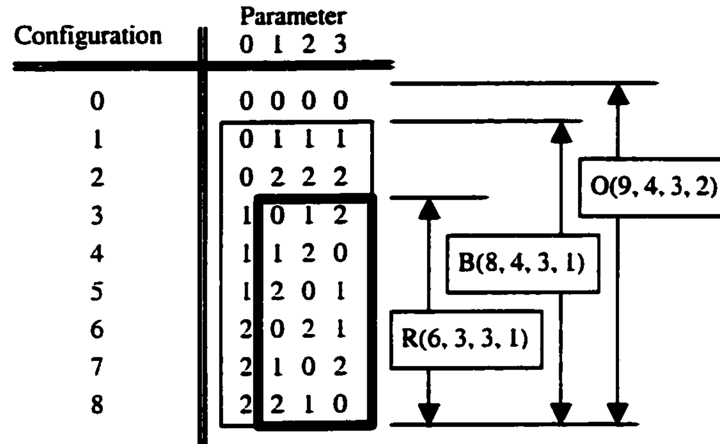


Figure 6-1: Orthogonal (O), basic (B), and reduced (R) arrays

There are two additional building blocks, not based on orthogonal arrays, that we will use for a general algorithm to construct covering arrays of strength 2:

Definition 6-3 Array of zeros, $Z(\rho, t)$

Let $Z(\rho, t)$ denote an array that contains all zeros, and is ρ rows by t columns. □

Definition 6-4 1 to $n-1$ array $N(n^2-n, n, t)$

Let $N(n^2-n, n, t)$ denote an array that contains a $n \times t$ block of ones, vertically concatenated with a $n \times t$ block of twos, and so on up to $n-1$. $N(n^2-n, n, t)$ has n^2-n rows and t columns. □

Examples of Z and N arrays are shown in Figure 6-2.

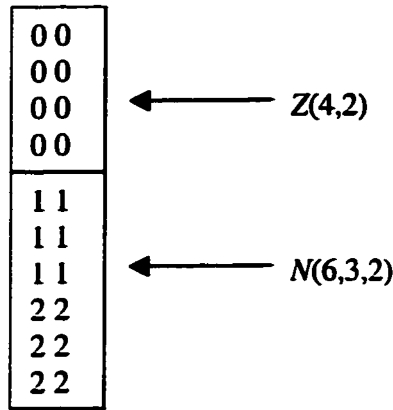


Figure 6-2: Examples of Z and N arrays

We will see how these building blocks are used in the next two sections.

6.2.2 Recursive covering array construction

In this section, we provide an illustration of an approach to constructing covering arrays of strength 2 that can handle more parameters than an orthogonal array. The precise algorithm will follow in section 6.3.

A "divide and conquer" approach can be taken. Suppose we want to construct a covering array for twelve parameters, each of which has three values. As a first stage, start by horizontally concatenating three copies of $O(9, 4, 3, 2)$, as shown in Figure 6-3. This results in an array with twelve parameters, and three values for each. The next step is to examine how close this array is toward achieving pair-wise coverage.

0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	1	1	1	0	1	1	1
0	2	2	2	0	2	2	2	0	2	2	2
1	0	1	2	1	0	1	2	1	0	1	2
1	1	2	0	1	1	2	0	1	1	2	0
1	2	0	1	1	2	0	1	1	2	0	1
2	0	2	1	2	0	2	1	2	0	2	1
2	1	0	2	2	1	0	2	2	1	0	2
2	2	1	0	2	2	1	0	2	2	1	0
🡅	✓	✓	✓	☒	✓	✓	✓	☒	✓	✓	✓

Figure 6-3: First step for constructing $C(15, 12, 3, 2)$

Consider the first column of the first copy of $O(9, 4, 3, 2)$. This is the column marked with a 🡅 in Figure 6-3. Because of the orthogonal array construction, pair-wise coverage is achieved between the first column and the remaining three columns in that first copy of $O(9, 4, 3, 2)$.

Each of the additional copies of $O(9, 4, 3, 2)$ has duplicates of those same three other columns, so pair-wise coverage has also been achieved with those columns as well. The first column has achieved pair-wise coverage with all columns marked with a ✓ in Figure 6-3.

The only column where pair-wise coverage has not yet been achieved against the first column is the leftmost column in each of the other copies of $O(9, 4, 3, 2)$. These columns are identical with the first column in the first copy of $O(9, 4, 3, 2)$. In these identical columns, only the pairs (0, 0), (1, 1), and (2, 2) are covered. Pair-wise coverage has not been achieved with the columns marked with an ☒ in Figure 6-3.

The strategy is then to cover the missing ordered pairs in a second stage. This is where the reduced array can be used. The array $R(6, 3, 3, 1)$ covers the non-(x, x) pairs for three parameters. Individual columns of $R(6, 3, 3, 1)$ can be inserted (as shown in Figure 6-4)

underneath the columns that have not yet achieved pair-wise coverage in our partially constructed array.

0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	1	1	1	0	1	1	1
0	2	2	2	0	2	2	2	0	2	2	2
1	0	1	2	1	0	1	2	1	0	1	2
1	1	2	0	1	1	2	0	1	1	2	0
1	2	0	1	1	2	0	1	1	2	0	1
2	0	2	1	2	0	2	1	2	0	2	1
2	1	0	2	2	1	0	2	2	1	0	2
2	2	1	0	2	2	1	0	2	2	1	0
0				1				2			
1				2				0			
2				0				1			
0				2				1			
1				0				2			
2				1				0			
↑				↑				↑			

Figure 6-4: Second step for constructing $C(15, 12, 3, 2)$

The extra rows have covered all the (x, x) combinations that were missed in the first step. Pair-wise coverage has now been achieved for the first column with all other columns. Since the leftmost column of each of the other orthogonal arrays is identical, pair-wise coverage has now been achieved for each column indicated by ↑ in Figure 6-4.

This process can be repeated for the rest of the columns in the first copy of $O(9, 4, 3, 2)$. In repeating the process, the first column of $R(6, 3, 3, 1)$ is used for each of the four columns in the first copy of $O(9, 4, 3, 2)$. The second column of $R(6, 3, 3, 1)$ is used for each of the four columns in the second copy of $O(9, 4, 3, 2)$, and so on. In effect, $R(6, 3, 3, 4)$ has been vertically concatenated below the three horizontally concatenated copies of $O(9, 4, 3, 2)$. The result is shown in Figure 6-5.

0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	1	1	1	0	1	1	1
0	2	2	2	0	2	2	2	0	2	2	2
1	0	1	2	1	0	1	2	1	0	1	2
1	1	2	0	1	1	2	0	1	1	2	0
1	2	0	1	1	2	0	1	1	2	0	1
2	0	2	1	2	0	2	1	2	0	2	1
2	1	0	2	2	1	0	2	2	1	0	2
2	2	1	0	2	2	1	0	2	2	1	0
0	0	0	0	1	1	1	1	2	2	2	2
1	1	1	1	2	2	2	2	0	0	0	0
2	2	2	2	0	0	0	0	1	1	1	1
0	0	0	0	2	2	2	2	1	1	1	1
1	1	1	1	0	0	0	0	2	2	2	2
2	2	2	2	1	1	1	1	0	0	0	0

Figure 6-5: The completed covering array $C(15, 12, 3, 2)$

Pair-wise coverage of all parameters has now been achieved. A "block composition" notation for this construction is introduced in Figure 6-6.

$O(9, 4, 3, 2)$	$O(9, 4, 3, 2)$	$O(9, 4, 3, 2)$
$R(6, 3, 3, 4)$		

Figure 6-6: $C(15, 12, 3, 2)$ in block composition notation

In the block composition notation, the contents of each building block array have been replaced by the array notation. It also shows how each building block is concatenated together. In particular, it shows that $R(6, 3, 3, 4)$ is concatenated under three copies of $O(9, 4, 3, 2)$, so that the width of the reduced array is shown in relation to the arrays above it.

This construction has two *stages*, where each "row" in the block composition notation represents a stage of the construction.

The selection of the reduced array is by the rule "use $R(n^2 - n, n, n, n + 1)$ under each group of n repetitions of $O(n^2, n + 1, n, 2)$." Up to $n^2 + n$ parameters can be handled using the reduced array.

If there are $n^2 + n < k \leq (n + 1)^2$ parameters, the reduced array cannot be used in the second stage, because the first stage will require $n + 1$ copies of $O(n^2, n + 1, n, 2)$ instead of n. Instead, the basic array $B(n^2 - 1, n + 1, n, n + 1)$ is used in the second stage. This is at the expense of $n - 1$ additional configurations. For example, if there are sixteen parameters, each with three values, we can construct $C(17, 16, 3, 2)$ as shown in Figure 6-7.

<table>
<tr><td>O(9, 4, 3, 2)</td><td>O(9, 4, 3, 2)</td><td>O(9, 4, 3, 2)</td><td>O(9, 4, 3, 2)</td></tr>
<tr><td colspan="4">B(8, 4, 3, 4)</td></tr>
</table>

Figure 6-7: $C(17, 16, 3, 2)$ in block composition notation

We could have used a copy of the orthogonal array $O(9, 4, 3, 2)$, with each column duplicated four times consecutively, in the second stage. However, this would result in a duplicate row containing entirely 0's, and this already appears in row 0. This is the motivation for using a basic array with the row of zeros removed.

If there are more than $(n + 1)^2$ parameters, the entire process can be repeated recursively in both the horizontal and vertical directions, as shown in Figure 6-8. In the following, O stands for $O(9, 4, 3, 2)$.

<table>
<tr><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td colspan="3">R(6, 3, 3, 4)</td><td colspan="3">R(6, 3, 3, 4)</td><td colspan="3">R(6, 3, 3, 4)</td></tr>
<tr><td colspan="9">R(6, 3, 3, 12)</td></tr>
</table>

Figure 6-8: $C(21, 36, 3, 2)$ in block composition notation

For each stage, the maximum number of parameters that can be accommodated is multiplied by $n + 1$, so the number of stages required is $s = \lceil \log_{n+1}(k) \rceil$.

6.2.3 Additional asymmetric columns

In this recursive process, an additional improvement can be made at each stage where the reduced arrays are used. We can do this because the addition of the extra configurations in the additional stage gives us a small increase in parameter capacity. Another effect of the algorithm to generate $O(n^2, n+1, n, 2)$ is that each column (except the first) is comprised of sets of permutations of $(0, \ldots, n-1)$. We can take advantage of this to increase the number of parameters at each stage.

To illustrate, create $C(15, 13, 3, 2)$ from $C(15, 12, 3, 2)$ by adding an additional asymmetric column (see Figure 6-9). The strategy is to put all zeros in the first n rows of the extra column next to the block of orthogonal arrays. For each of the $n-1$ groups of n rows next to the reduced array, put n copies of each of the values 1 through $n-1$ in the extra column.

0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	1	1	1	0	1	1	1	0
0	2	2	2	0	2	2	2	0	2	2	2	0
1	0	1	2	1	0	1	2	1	0	1	2	
1	1	2	0	1	1	2	0	1	1	2	0	
1	2	0	1	1	2	0	1	1	2	0	1	
2	0	2	1	2	0	2	1	2	0	2	1	
2	1	0	2	2	1	0	2	2	1	0	2	
2	2	1	0	2	2	1	0	2	2	1	0	
0	0	0	0	1	1	1	1	2	2	2	2	1
1	1	1	1	2	2	2	2	0	0	0	0	1
2	2	2	2	0	0	0	0	1	1	1	1	1
0	0	0	0	2	2	2	2	1	1	1	1	2
1	1	1	1	0	0	0	0	2	2	2	2	2
2	2	2	2	1	1	1	1	0	0	0	0	2
⊠	✓	✓	✓	⊠	✓	✓	✓	⊠	✓	✓	✓	↑

Figure 6-9: First step of constructing $C(15, 13, 3, 2)$

In Figure 6-9, the asymmetric column is marked with a ↑. Since every column in the R array consists of permutations of (0, 1, 2), and each column (except the left) of the orthogonal array consists of permutations of (0, 1, 2), we have achieved pair-wise coverage between the asymmetric column and all columns marked with a ✓.

Because of the exception to the permutation property, the extra column does not have pair-wise coverage with any of the left-most columns of the orthogonal arrays (those marked with an ⊠ in Figure 6-9). For those columns, we are missing the (1, 0) and (2, 0) combinations. These can be covered by filling in two of the six unfilled positions, as shown in Figure 6-10.

0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	0	1	1	1	0	1	1	1	0
0	2	2	2	0	2	2	2	0	2	2	2	0
1	0	1	2	1	0	1	2	1	0	1	2	0
1	1	2	0	1	1	2	0	1	1	2	0	–
1	2	0	1	1	2	0	1	1	2	0	1	–
2	0	2	1	2	0	2	1	2	0	2	1	0
2	1	0	2	2	1	0	2	2	1	0	2	–
2	2	1	0	2	2	1	0	2	2	1	0	–
0	0	0	0	1	1	1	1	2	2	2	2	1
1	1	1	1	2	2	2	2	0	0	0	0	1
2	2	2	2	0	0	0	0	1	1	1	1	1
0	0	0	0	2	2	2	2	1	1	1	1	2
1	1	1	1	0	0	0	0	2	2	2	2	2
2	2	2	2	1	1	1	1	0	0	0	0	2

Figure 6-10: The completed array $C(15, 13, 3, 2)$

The remaining four positions are left as "don't care" values, which means that for that particular test configuration, any value can be used.

Because this step results in "don't care" values, and an unbalanced use of values (the "0" in the example), the tester can judiciously choose the correspondence between the

enumerated values 0, 1, ..., $n - 1$, and actual system configuration values. For example, if a particular system element has not been as thoroughly tested in the unit testing phase, the system tester could assign this element to correspond to "0" in the extra column, and fill in the "don't care" values with zeros as well. This will result in this particular system element being used in more test configurations. For the rest of this section, the don't care values will be arbitrarily filled in as zeros, to form a column of n^2 zeros. However, it is useful to remember that there are some entries where a zero is not strictly required.

A notation for this construction of $C(15, 13, 3, 2)$ in Figure 6-10 is shown in Figure 6-11.

<table>
<tr><td>$O(9, 4, 3)$</td><td>$O(9, 4, 3)$</td><td>$O(9, 4, 3)$</td><td>$Z(9, 1)$</td></tr>
<tr><td colspan="3">$R(6, 3, 3, 4)$</td><td>$N(6, 3, 1)$</td></tr>
</table>

Figure 6-11: $C(15, 13, 3, 2)$ in block composition notation

Note that we cannot use this asymmetric column approach in stages where the basic array is used instead of the reduced array. The first t columns of the basic array would, because of the permutation exception, be identical to those in the N array, and therefore pair-wise coverage would not be achieved.

For a third stage, when copies are made horizontally, the extra columns are included in the copies. In Figure 6-12, the results of the second stage are copied horizontally three times. To keep the extra columns together, they are denoted as $Z(9, 3)$ and $N(6, 3, 3)$. Pair-wise coverage must be ensured among these extra columns, and that is achieved by concatenating $R(6, 3, 3, 3)$ vertically below these columns. Because a reduced array, $R(6, 3, 3, 12)$ is used in the third stage, we can also add an extra column, $Z(15, 1)$ and $N(6, 3, 1)$ for that stage.

<table>
<tr><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>$Z(9, 3)$</td><td rowspan="2">$Z(15, 1)$</td></tr>
<tr><td colspan="3">$R(6, 3, 3, 4)$</td><td colspan="3">$R(6, 3, 3, 4)$</td><td colspan="3">$R(6, 3, 3, 4)$</td><td>$N(6, 3, 3)$</td></tr>
<tr><td colspan="9">$R(6, 3, 3, 12)$</td><td>$R(6, 3, 3, 3)$</td><td>$N(6, 3, 1)$</td></tr>
</table>

Figure 6-12: $C(21, 40, 3, 2)$ in block composition notation

6.3 Building the covering array

The building blocks defined in section 6.2.1 give us the elements that are needed to construct $C(\rho, k, n, 2)$ according to the guidelines in sections 6.2.2 and 6.2.3. This section describes how to assemble the building blocks.

Definition 6-5: Number of stages, s

Let s denote the number of stages in the construction of the covering array. □

We have already presented an intuitive argument in section 6.2.2 that the number of stages required is $s = \lceil \log_{n+1}(k) \rceil$, but for now, we can assume that s is a finite positive integer. A formal proof of this result will follow as Lemma 6-9.

If $k \leq n + 1$, then a subset of k columns of $O(n^2, n + 1, n, 2)$ can be used as $C(\rho, k, n, 2)$, and it is not necessary to proceed further.

Definition 6-6 Number of columns to use in array building blocks, g_r

Let g_r represent the number of columns to be used in the arrays at stage r. □

For the initial stage, $g_0 = n + 1$, as the arrays $O(n^2, n + 1, n, 2)$ are always used. For subsequent stages, $g_r = n + 1$ indicates that at stage r, copies of the basic (B) array are used. If $g_r = n$, then copies of the reduced (R) array are used.

The initial number of columns for arrays in each stage is $g_0 = n + 1$ (representing the use of the orthogonal array $O(n^2, n + 1, n, 2)$ in the initial stage), and $g_r = n$ for $r = 1, \ldots, s - 1$. We are hoping to use reduced arrays for all stages after the initial one, to keep the number of additional test configurations to a minimum. However, when the number of parameters necessitates the use of a basic array, g_r will be set to $n + 1$.

Definition 6-7 Parameter capacity at stage r, h_r

Let h_r represent the total parameter capacity at stage r. □

To calculate the values of g_r, an algorithm to determine the parameter capacity will be used as a subroutine:

Algorithm 6-1 Determine parameter capacity

Parameter capacity $h_0 \leftarrow n + 1$
For $r = 1, \ldots, s - 1$:
  $h_r \leftarrow h_{r-1} \times g_r$
  If $g_r = n$ then
    $h_r \leftarrow h_r + 1$ (for extra column)
  End if
End for
□

The algorithm works as follows. With an orthogonal array in the initial stage, we have the capacity for $n + 1$ parameters. For each stage that is added, we can increase the number of orthogonal arrays used by either n or $n + 1$, namely, the value of g_r. When we have the case where reduced arrays are used (that is, $g_r = n$), we can add one more column for the asymmetric column. The result is the number of parameters h that can be handled for a given number of stages s, and for a set of values g_r. This leads to how we can determine the set of values g_r.

Algorithm 6-2 Determine usage of basic versus reduced arrays

$g_0 \leftarrow n + 1$, and $g_r \leftarrow n$ for $r = 1, \ldots, s - 1$
$r \leftarrow 1$.
Determine parameter capacity h.
While $h_{s-1} < k$:
  $g_r \leftarrow n + 1$
  $r \leftarrow r + 1$
  Recalculate h_{s-1}
End while
□

After using Algorithm 6-2, $g_r = n + 1$ indicates that at stage r, copies of the basic array are used. If $g_r = n$, then copies of the reduced array are used at stage r. The strategy is to start with $g_r = n$ for all values of $r > 0$. This would represent using reduced arrays at every stage after the initial stage. The rationale is that reduced arrays add fewer configurations than the basic arrays, and so we should use them wherever possible. However, if this does not provide enough parameter capacity for the value of k, then

some of the values for g_r must be increased. We choose to increase g_1 to $n + 1$ first, as this gives us the largest additional parameter capacity for the expense of adding $n - 1$ configurations. The approach is to increase the values of g_r in order, to handle the most parameters in the fewest number of configurations.

The next calculation is the values to use for t in the basic or reduced arrays at each stage.

Definition 6-8 Column duplication values, t_r

Let t_r represent the column duplication value at stage r. Let $t_0 = 1$, and $t_r = t_{r-1} \times g_{r-1}$ for $r = 1, \ldots, s - 1$. □

The rationale here is that when a basic or reduced array is used, the columns have to be duplicated as many times as the width of the arrays used in the previous stage. This is represented by the number of columns in the array used at the previous stage, multiplied by the number of times those columns were duplicated at the previous stage.

Lemma 6-3 Value of t_r

Let t_r be determined as in Definition 6-8. Then $t_r = \prod_{b=0}^{r-1} g_b$.

Proof (by induction)

Basis: From Definition 6-8, we have $t_0 = 1 = \prod_{b=0}^{0-1} g_b$, using our assumption about products with zero terms.

Induction step: assume $t_r = \prod_{b=0}^{r-1} g_b$. By Definition 6-8, we have $t_{r+1} = t_r \times g_r$ $= \prod_{b=0}^{r-1} g_b \times g_r = \prod_{b=0}^{r} g_b$.

Therefore, by induction, $t_r = \prod_{b=0}^{r-1} g_b$. □

Not surprisingly, the result $t_r = \prod_{b=0}^{r-1} g_b$ indicates that at any stage, the columns are duplicated by a factor representing the product of the width of the arrays used in previous stages.

For each stage that uses reduced arrays, we need to determine the parameters for the extra columns.

Definition 6-9 Number of asymmetric columns, $e_r(s)$

Let $e_r(s)$ be the number of extra columns in stage r, when there are s stages in total. Thus, at stage r, the arrays $Z(\rho, e_r(s))$ and $N(n^2 - n, n, e_r(s))$ are used, where ρ is the current number of rows in the covering array.

Let: $e_r(s) = \begin{cases} 0, \text{if } g_r = n+1 \\ t_{s-1} g_{s-1} / t_r g_r, \text{if } g_r = n \end{cases}$. □

The value of $e_r(s)$ is only non-zero for stages where reduced arrays are used; that is, when $g_r = n$. No asymmetric columns are added when basic or orthogonal arrays are used at a particular stage.

At this point, there is a detail to address: is the value of $e_r(s)$ always an integer? This will be determined in the following lemma:

Lemma 6-4 Value of $e_r(s)$

Let t_r be defined as in Definition 6-9. Then the value of $t_{s-1}\, g_{s-1} / t_r\, g_r$ is always an integer, and $e_r(s) = \prod_{b=r}^{s-1} g_b$.

Proof:

Clearly, if $g_r = n + 1$, then $e_r(s) = 0$ is an integer. We need to show that the value $t_{s-1}\, g_{s-1} / t_r\, g_r$ is also an integer for all r.

By Lemma 6-3, $t_r = \prod_{b=0}^{r-1} g_b$, so $t_{s-1}\, g_{s-1} / t_r\, g_r = (\prod_{b=0}^{s-2} g_b \times g_{s-1}) / (\prod_{b=0}^{r-2} g_b \times g_{r-1})$

$= \prod_{b=0}^{s-1} g_b / \prod_{b=0}^{r-1} g_b = \prod_{b=r}^{s-1} g_b$, which is an integer. □

The reason why the number of asymmetric columns at any stage is dependent on both r and s is as follows. At each stage when we have $g_r = n$, we can add one extra asymmetric column. These columns are duplicated each time there is a subsequent stage. Therefore, the total number of columns associated with a particular stage r is dependent on the number of remaining stages, namely the difference between r and s. In particular, the columns are duplicated by the product of the width of subsequent building blocks added, namely $\prod_{b=r}^{s-1} g_b$.

Finally, there is the duplication factor for the reduced arrays that are added for the extra columns. The set of extra columns added at each stage is treated differently at each subsequent stage.

Definition 6-10 Duplication factor for asymmetric columns

Let λ_{qr} be the duplication factor for the reduced array added at stage r, for the extra columns added at stage q. For the columns added at any particular stage q, the duplication factor starts at one in the next stage, and is multiplied by g_{r-1} for subsequent stages r.

Let: $\lambda_{qr} \leftarrow \begin{cases} 1, \text{if } r = q+1 \text{ and } q+1 < s \\ \lambda_{q,r-1} \times g_{r-1} \text{ for } r = q+2,\ldots,s-1 \end{cases}$ □

Note that while it is required for $g_r = n$ for $e_r(s)$ to be non-zero, the duplication factor for subsequent stages is just increased by the value of g_r no matter what value it has. If we follow Algorithm 6-2, once $g_r = n$ for some stage, $g_r = n$ for all subsequent stages. However, we are keeping this definition general in case we decide to allow full flexibility of choosing the values of g_r. This will be discussed further in section 9.3.3.

We shall now determine the value of λ_{qr}.

Lemma 6-5 Value of λ_{qr}

Suppose q is fixed such that $0 < q < s - 1$. Then for values of $r > q$, $\lambda_{qr} = \prod_{b=q+1}^{r-1} g_b$.

Proof (by induction):

Basis: $r = q + 1$.

By Definition 6-10, we have $\lambda_{qr} = 1$, when $r = q + 1$. Thus, $\lambda_{q,q+1} = 1 = \prod_{b=q+1}^{q} g_b$ (using our assumption about products with no terms).

Induction:

Suppose that $\lambda_{qr} = \prod_{b=q+1}^{r-1} g_b$.

By Definition 6-10, $\lambda_{q,r+1} = \lambda_{qr} \times g_r$. Therefore, we have:

$$\lambda_{q,r+1} = \lambda_{qr} \times g_r = \prod_{b=q+1}^{r-1} g_b \times g_r = \prod_{b=q+1}^{r} g_b .$$

Therefore, by induction, $\lambda_{qr} = \prod_{b=q+1}^{r-1} g_b$. □

We now turn to the complete construction algorithm. The following notation is used:

- C is the covering array to be constructed
- S is the array to be added at the current stage.
- A is a building block to be added
- $\varnothing$ is an empty array
- $X \blacktriangleleft Y$ means "concatenate array Y horizontally to the right of X"

- $X \blacktriangledown Y$ means "concatenate array Y vertically below X"

It is also assumed that if an array has a duplication factor of zero, it is a null array, and concatenating a null array results in the original array.

Algorithm 6-3 Construct $C(\rho, k, n, 2)$

The following.algorithm builds $C(\rho, k, n, 2)$.

$C \leftarrow \varnothing$
For $r = 0, \ldots, s-1$:
  $C \leftarrow C \blacktriangleleft Z(\rho, e_r(s))$; ρ is current number of rows in C.

$$A \leftarrow \begin{cases} O(n^2, n+1, n, 2), & \text{if } r = 0 \\ R(n^2 - n, n, n, t_r), & \text{if } r > 0, g_r = n \\ B(n^2 - 1, n+1, n, t_r), & \text{if } r > 0, g_r = n+1 \end{cases}$$

  $S \leftarrow \varnothing$
  $S \leftarrow S \blacktriangleleft A$, repeated $(t_{s-1}\, g_{s-1}) / (t_r\, g_r)$ times
  For $q = 1, \ldots, r-1$:

$$A \leftarrow \begin{cases} R(n^2 - n, n, n, \lambda_{qr}), & \text{if } g_r = n \\ B(n^2 - 1, n+1, n, \lambda_{qr}), & \text{if } g_r = n+1 \end{cases}$$

    $S \leftarrow S \blacktriangleleft A$, repeated $e_q(s) / (\lambda_{qr}\, g_r)$ times
  End for
  $S \leftarrow S \blacktriangleleft N(n^2 - n, n, e_r(s))$
  $C \leftarrow C \blacktriangledown S$
End for
□

For the first stage, the algorithm horizontally concatenates the required number of orthogonal arrays. Starting with the second stage, an extra column Z array is horizontally concatenated, if required. Then a new stage array is started. Depending on what is required, the proper number of basic or reduced arrays is horizontally concatenated to the stage array. Then reduced arrays for extra columns are horizontally concatenated to the new stage array. Next, the N array is horizontally concatenated to the new stage array. Finally, the new stage array is vertically concatenated onto the overall set of configurations.

6.4 Proof that the algorithm results in a covering array

In this section, we show that the result of Algorithm 6-3 is a covering array of a sufficiently large size in terms of the number of parameters.

The overall strategy to prove that Algorithm 6-3 is correct is as follows:

- Show that all the vertical and horizontal array concatenations used in the algorithm are well defined.
- Show that the number of columns in the array is at least k.
- Show that the number of stages is sufficient.
- Choose two columns from the array, and show that all pair-wise combinations of values for the selected parameters are covered within the two columns.

First, we shall prove a number of lemmas with respect to the vertical and horizontal array concatenations. This will show that they are well defined, and give us some results for the dimensions of the concatenated arrays. We shall start with the innermost loop inside Algorithm 6-3, and work outwards.

Lemma 6-6 Columns in S after concatenations of A

Let $C(\rho, k, n, 2)$ be constructed using Algorithm 6-3. In this algorithm, we have the section:

$$A \leftarrow \begin{cases} O(n^2, n+1, n, 2), \textit{ if } r=0 \\ R(n^2-n, n, n, t_r), \textit{ if } r>0,\ g_r = n \\ B(n^2-1, n+1, n, t_r), \textit{ if } r>0,\ g_r = n+1 \end{cases}$$

$$S \leftarrow \varnothing$$

$S \leftarrow S \blacktriangleleft A$, repeated $(t_{s-1}\, g_{s-1}) / (t_r\, g_r)$ times

Then for $0 \le r < s$, the concatenation $S \leftarrow S \blacktriangleleft A$ is well defined, and after all repetitions, S will have $\prod_{b=0}^{s-1} g_b$ columns.

Proof:

First, we note that by Lemma 6-4, $(t_{s-1}\, g_{s-1}) / (t_r\, g_r)$ is always an integer.

For $r = 0$, from Algorithm 6-3, $(t_{s-1}\, g_{s-1}) / (t_r\, g_r)$ copies of $A = O(n^2, n + 1, n, 2)$ are concatenated horizontally. For $r = 0$, from Definition 6-8, we have $t_0 = 1$, and from Algorithm 6-2, $g_0 = n + 1$. Therefore, the horizontal concatenation is repeated $(t_{s-1}\, g_{s-1}) / (n + 1)$ times. However, the array $O(n^2, n + 1, n, 2)$ has width $n + 1$, so the total width of the concatenated orthogonal arrays S is $t_{s-1}\, g_{s-1}$. The operation is well defined since the array O is constant, and always has the same number of rows.

For $r > 0$, from Algorithm 6-3, $(t_{s-1}\, g_{s-1}) / (t_r\, g_r)$ copies of A are concatenated horizontally. The operation is well defined since the array A is constant, and always has the same number of rows. The number of columns in A (for either construction from the B or R arrays) is g_r, and each column in A is duplicated t_r times. Therefore, the number of columns in A is $t_r\, g_r$. Therefore, the number of columns in S is $(t_{s-1}\, g_{s-1}) / (t_r\, g_r) \times (t_r\, g_r) = t_{s-1}\, g_{s-1}$.

Therefore, for $0 \le r < s$, S will have $t_{s-1}\, g_{s-1}$ columns. By Lemma 6-3, $t_{s-1}\, g_{s-1} = \prod_{b=0}^{s-1} g_b$ □

Lemma 6-7 Columns in *S* after concatenating *R* and *N* arrays

Let $C(\rho, k, n, 2)$ be constructed using Algorithm 6-3. In this algorithm, we have the following section:

(algorithm fragment from Lemma 6-6)

For $q = 1, \ldots, r-1$:

$$A \leftarrow \begin{cases} R(n^2-n,n,n,\lambda_{qr}), \text{ if } g_r = n \\ B(n^2-1,n+1,n,\lambda_{qr}), \text{ if } g_r = n+1 \end{cases}$$

$S \leftarrow S$ ◀ A, repeated $e_q(s) / (\lambda_{qr}\, g_r)$ times

End for

$S \leftarrow S$ ◀ $N(n^2 - n, n, e_r(s))$

Then for $0 \le r < s$, at the end of this section, S will have $\prod_{b=0}^{s-1} g_b + \sum_{q=0}^{r} e_q(s)$ columns and the two horizontal concatenation operations for the R and N arrays are well defined.

Proof:

For $r = 0$, from Definition 6-9, we have $e_0(s) = 0$. By Lemma 6-6, S will have $\prod_{b=0}^{s-1} g_b$ columns at the start of the loop indexed by q. Since this loop runs from 1 to $r-1$, it will not be executed when $r = 0$. Furthermore, since $e_0(s) = 0$, $N(n^2 - n, n, e_0(s))$ is a null array. Therefore, the concatenation of N is well defined, since concatenation of a null array leaves the original array. We have $\prod_{b=0}^{s-1} g_b + e_0(s)$ columns since no columns were added, and $e_0(s) = 0$.

For $r > 0$, by Lemma 6-6, S will have $\prod_{b=0}^{s-1} g_b$ columns at the start of the loop indexed by q. Inside the loop indexed by q, one of the arrays $R(n^2 - n, n, n, \lambda_{qr})$ or $B(n^2 - 1, n + 1, n, \lambda_{qr})$ is horizontally concatenated $e_q(s) / \lambda_{qr}\, g_r$ times. However, the number of columns in either array is $\lambda_{qr}\, g_r$, so a total of $e_q(s)$ columns are added. For the entire iteration of the loop, $\sum_{q=0}^{r-1} e_q(s)$ columns are added. The loop iterates from 1 to $r - 1$, but since $e_0(s) = 0$, we can index the summation from 0 to $r - 1$. After this loop is completed, the array

$N(n^2 - n, n, e_r(s))$ is horizontally concatenated. The N array has $e_r(s)$ columns, so

$$e_r(s) + \sum_{q=0}^{r-1} e_q(s) = \sum_{q=0}^{r} e_q(s).$$

Each array $R(n^2 - n, n, n, \lambda_{qr})$ that is concatenated has $n^2 - n$ rows. By the construction of A, S will have $n^2 - n$ rows if the R array is used, and $n^2 - 1$ rows if the B array is used. However, from Definition 6-9, when $g_r = n + 1$, $e_r(s) = 0$. Thus, when the B array is used, $e_r(s) = 0$. The horizontal concatenation in the loop indexed by q is repeated zero times, and therefore is not performed. Furthermore, $N(n^2 - n, n, e_r(s))$ is a null array. Thus, the concatenation of N is well defined, since concatenation of a null array leaves the original array. Therefore, when $g_r = n + 1$, the two horizontal concatenations are well defined.

When $g_r = n$, $e_r(s) \neq 0$, and the R array is used to form A. Thus, the array S has $n^2 - n$ rows at the start of the loop indexed by q. By Lemma 6-6, the number of columns in S is $\prod_{b=0}^{s-1} g_b$ at the start of the loop indexed by q. Therefore, the repeated horizontal concatenation of $R(n^2 - n, n, n, \lambda_{rq})$ is well defined, since this array has $n^2 - n$ rows. The array $N(n^2 - n, n, e_r(s))$ also has $n^2 - n$ rows, so the horizontal concatenation of this array is well defined.

Therefore, for $0 \leq r < s$, S will have $\prod_{b=0}^{s-1} g_b + \sum_{q=1}^{r} e_q(s)$ columns and the two horizontal concatenation operations for the R and N arrays are well defined.□

Lemma 6-8 Concatenations involving array C, and number of columns in C

Let $C(\rho, k, n, 2)$ be constructed using Algorithm 6-3. In this algorithm, we have the following fragment:

$C \leftarrow \varnothing$
For $r = 0, \ldots, s-1$:
    $C \leftarrow C \blacktriangleleft Z(\rho, e_r(s))$; ρ is current number of rows in C.

    (algorithm fragment from Lemma 6-7— no modifications to C)

    $C \leftarrow C \blacktriangledown S$
End for

Then each of the above concatenation operations is well defined, and the array C has $\prod_{b=0}^{s-1} g_b + \sum_{q=0}^{r} e_q(s)$ columns.

Proof: (by induction)

Basis: For $r = 0$, from Definition 6-9, we have $e_0(s) = 0$. Therefore, $Z(\rho, e_0(s))$ is a null array, and the concatenation operation is well defined since we are concatenating two null arrays. Furthermore, since C is still null for the vertical concatenation of S, this operation is also well defined. At the end of the loop, C has $\prod_{b=0}^{s-1} g_b + e_0(s)$ columns by Lemma 6-7.

Induction step: For $r > 0$, assume that C has $\prod_{b=0}^{s-1} g_b + \sum_{q=0}^{r-1} e_q(s)$ columns after iteration $r-1$. The horizontal concatenation of C and $Z(\rho, e_r(s))$ is well defined, because Z is defined to have ρ rows, and ρ is the current number of rows in C. Before this horizontal concatenation, C has $\prod_{b=0}^{s-1} g_b + \sum_{q=0}^{r-1} e_q(s)$ columns by the inductive assumption. After the horizontal concatenation, C has $\prod_{b=0}^{s-1} g_b + \sum_{q=0}^{r-1} e_q(s) + e_r(s) = \prod_{b=0}^{s-1} g_b + \sum_{q=0}^{r} e_q(s)$ columns. By Lemma 6-7, the array S will

also have $\prod_{b=0}^{s-1} g_b + \sum_{q=0}^{r} e_q(s)$ columns. Thus, the arrays C and S will have the same number of columns, and therefore, the vertical concatenation is well defined.

Therefore, for $0 \le r < s$, the two concatenation operations for the Z and C arrays are well defined, and the number of columns in C is $\prod_{b=0}^{s-1} g_b + \sum_{q=0}^{r} e_q(s)$. □

At this point, we have verified that all the array concatenations used within Algorithm 6-3 are well-defined. We also have results for the number of columns that result from these operations.

Next, we shall verify that the number of stages required is sufficient. That is, we shall make sure that the outer loop of Algorithm 6-3 iterates the correct number of times.

Lemma 6-9 Number of stages required

The number of stages required is $s = \lceil \log_{n+1}(k) \rceil$.

Proof:

By Lemma 6-8, the number of columns in C is $\prod_{b=0}^{s-1} g_b + \sum_{q=0}^{r} e_q(s)$. By Definition 6-6, the maximum number of columns to be used at any stage is $n + 1$, and whenever $g_r = n + 1$, $e_r(s) = 0$. Therefore, after stage $s - 1$, the maximum number of columns is $\prod_{b=0}^{s-1} g_b = (n + 1)^s$. We want $k \le (n + 1)^s$, so $\log_{n+1}(k) \le s$. Since s should be an integer, we have $s = \lceil \log_{n+1}(k) \rceil$. □

We are now ready to show that the array generated by Algorithm 6-3 has at least k columns. That is, it has values for all of the parameters.

Lemma 6-10 The array C has a sufficient number of columns

Let C be constructed using Algorithm 6-3. Then C has at least k columns.

Proof:

By Lemma 6-8, after s iterations of the outer loop in Algorithm 6-3, the number of columns in C is $\prod_{b=0}^{s-1} g_b + \sum_{b=0}^{s-1} e_b(s)$. Therefore, we need to show that $k \le \prod_{b=0}^{s-1} g_b + \sum_{b=0}^{s-1} e_b(s)$.

By Lemma 6-9, we have $s = \lceil \log_{n+1}(k) \rceil$, and therefore, $k \le (n+1)^s$. The goal is to show two properties in Algorithm 6-2. The first property is that for h_{s-1} calculated by Algorithm 6-1, $h_{s-1} = \prod_{b=0}^{s-1} g_b + \sum_{b=0}^{s-1} e_b(s)$. The second property is that the while loop in Algorithm 6-2 will terminate after at most s iterations.

Show $h_{s-1} = \prod_{b=0}^{s-1} g_b + \sum_{b=0}^{s-1} e_b(s)$ by induction on s.

Base: $s = 1$: In Algorithm 6-1, we start with $h_0 = n + 1$. Since $g_0 = n + 1$, we have $e_0(1) = 0$. Therefore, $h_0 = g_0 + e_0(1)$.

Inductive assumption: $h_{s-1} = \prod_{b=0}^{s-1} g_b + \sum_{b=0}^{s-1} e_b(s)$.

Induction: In Algorithm 6-1, we first have the step $h_s = h_{s-1} \times g_s$. Therefore, we have $h_s = g_s \times \prod_{b=0}^{s-1} g_b + g_s \times \sum_{b=0}^{s-1} e_b(s)$. Now $g_s \times \sum_{b=0}^{s-1} e_b(s) = \sum_{b=0}^{s-1} \frac{t_{s-1} g_{s-1} g_s}{t_b g_b}$. By Lemma 6-3, we have $t_s = t_{s-1} \times g_{s-1}$, so $g_s \times \sum_{b=0}^{s-1} e_b(s) = \sum_{b=0}^{s-1} \frac{t_s g_s}{t_b g_b} = \sum_{b=0}^{s-1} e_b(s+1)$.

Therefore, we have $h_s = \prod_{b=0}^{s} g_b + \sum_{b=0}^{s-1} e_b(s+1)$. The next step in Algorithm 6-1 is to add 1, if $g_s = n$. If we have $g_s = n + 1$ instead, then $e_s(s+1) = 0$. If we have $g_s = n$,

then $e_s(s+1) = \frac{t_s g_s}{t_s g_s} = 1$. Thus, no matter what the value of g_{s+1}, we are adding $e_s(s+1)$. We now have $h_s = \prod_{b=0}^{s} g_b + \sum_{b=0}^{s-1} e_b(s+1) + e_s(s+1) = \prod_{b=0}^{s} g_b + \sum_{b=0}^{s} e_b(s+1)$.

Therefore, by induction on s, $h_{s-1} = \prod_{b=0}^{s-1} g_b + \sum_{b=0}^{s-1} e_b(s)$.

The remainder is straightforward. If Algorithm 6-2 terminates in fewer than s iterations, we have $h \geq k$. If we reach iteration number $s - 1$, this means that we have set $g_r = n + 1$, for r from $1, \ldots, s - 1$. We already have $g_0 = n + 1$. Therefore, $g_r = n + 1$ for all r from $1, \ldots, s - 1$. Therefore, $h_{s-1} = \prod_{b=0}^{s-1} g_b + \sum_{b=0}^{s-1} e_b(s)$ $= (n + 1)^s + 0 = (n + 1)^s$; but we have $k \leq (n + 1)^s$, so Algorithm 6-2 must terminate after iteration $s - 1$.

We have shown that Algorithm 6-2 will always terminate with $h \geq k$, so we have established that $k \leq \prod_{b=0}^{s-1} g_b + \sum_{b=0}^{s-1} e_b(s)$. By Lemma 6-8, the number of columns in the covering array C is $\prod_{b=0}^{s-1} g_b + \sum_{b=0}^{s-1} e_b(s)$, so therefore, the number of columns in C is at least k.

Therefore, the covering array contains a sufficient number of columns. □

We now introduce a lemma that is needed for the proof that C is a covering array. The lemma shows that if two numbers are congruent modulo m_1, but not congruent modulo $m_1 m_2$, then their difference modulo $m_1 m_2$ is at least m_1. This will be applied to the column indices of two columns we will be comparing for pair-wise coverage. If the two indices represent identical columns in arrays used at one stage, but not in the next stage, we want to show that they are sufficiently far apart that the difference between the column indices is greater than the column duplication factor used at that stage.

Lemma 6-11 Results when modulus increases

Suppose that i, and j are integers ≥ 0, m_1 and m_2 are integers > 0, and $j > i$. If $i \bmod m_1 = j \bmod m_1$, and $i \bmod m_1 m_2 \neq j \bmod m_1 m_2$, then: $| i \bmod m_1 m_2 - j \bmod m_1 m_2 | \geq m_1$.

Proof:

Let $a = i \bmod m_1m_2$, and $b = j \bmod m_1m_2$. Then for some integers k_1, $k_2 \geq 0$, we have $i = k_1m_1m_2 + a$ and $j = k_2m_1m_2 + b$, where $b \neq a$. Therefore, $b - a = j - i + (k_1 - k_2)m_1m_2$. Let $c = i \bmod m_1 = j \bmod m_1$, so $i = k_3m_1 + c$, and $j = k_4m_1 + c$ for some integers k_3, $k_4 \geq 0$. Therefore, $j - i = (k_4 - k_3)m_1$.

Thus, we have $b - a = (k_4 - k_3)m_1 - (k_1 - k_2)m_1m_2$. We have $b \neq a$, so $(k_4 - k_3)m_1 - (k_1 - k_2)m_1m_2 \neq 0$. Therefore, $((k_4 - k_3) + (k_1 - k_2)m_2)m_1 \neq 0$. We have $m_1 > 0$, so we must also have $(k_4 - k_3) + (k_1 - k_2)m_2 \neq 0$. But, $b - a = km_1$, for some non-zero integer k, so $| b - a | \geq m_1$.

Thus, $| i \bmod m_1 m_2 - j \bmod m_1 m_2 | \geq m_1$. □

At this point, we have verified that the dimensions of the generated array are correct, and that all array concatenations used in Algorithm 6-3 are well defined. The last step is to verify that C has the properties of a covering array. That is, if we select any two columns, every pair-wise combination of parameter values occurs at least once in the set of ordered pairs formed by the two columns.

Theorem 6-1 The array C is a covering array of strength 2

Let $C(\rho, k, n, 2)$ be constructed using Algorithm 6-3. Then $C(\rho, k, n, 2)$ satisfies the required property of a covering array. That is, select columns i, j of $C(\rho, k, n, 2)$ such that $0 \leq i, j < k$, and $i \neq j$. Then all pair-wise combinations between those columns are covered.

Proof:

We make a distinction between the "regular" columns and the "extra" columns, where the regular columns are from 0 to $\left(\prod_{b=0}^{s-1} g_b\right)-1$, and the extra columns are from $\prod_{b=0}^{s-1} g_b$ to $\prod_{b=0}^{s-1} g_b + \sum_{b=0}^{s-1} e_b(s)$. This distinction is due to the various sorts of building blocks we have used to construct the array.

Suppose that we select two columns, i and j, from the array C, where $i \neq j$. Without loss of generality, assume that $i < j$.

Depending on the particular columns selected, we could have the following cases:

Case 1: Both columns are "regular." That is, $i, j < \prod_{b=0}^{s-1} g_b$.

Case 2: One column is "regular" and one column is "extra". That is, $i < \prod_{b=0}^{s-1} g_b$ and $j \geq \prod_{b=0}^{s-1} g_b$.

Case 3: Both columns are "extra". That is, $i, j \geq \prod_{b=0}^{s-1} g_b$.

We now examine each of these cases:

Case 1: Both columns are "regular."

In this case, we have $i, j < \prod_{b=0}^{s-1} g_b$.

We first examine whether the two columns represent the same or differing columns in the orthogonal arrays. Since we have used copies of $O(n^2, n + 1, n, 2)$, the orthogonal arrays are always $n + 1$ columns in width. Check if

$i \bmod (n + 1) \neq j \bmod (n + 1)$. If this is true, call this sub-case 1.1. Otherwise, if $i \bmod (n + 1) = j \bmod (n + 1)$, call this sub-case 1.2.

Sub-case 1.1:

The interpretation of $i \bmod (n + 1) \neq j \bmod (n + 1)$ is that the two columns correspond to different columns in the array $O(n^2, n + 1, n, 2)$, namely columns $i \bmod (n + 1)$, and $j \bmod (n + 1)$. By the construction of $O(n^2, n + 1, n, 2)$, all pair-wise combinations are covered within two different columns of $O(n^2, n + 1, n, 2)$. Therefore, all pair-wise combinations between columns i and j are covered within the first n^2 rows of the covering array. We can terminate this branch of our sub cases.

Sub-case 1.2

The interpretation of $i \bmod (n + 1) = j \bmod (n + 1)$ is that the two columns correspond to identical columns in the array $O(n^2, n + 1, n, 2)$. Therefore, all of the combinations $(0, 0), (1, 1), \ldots, (n - 1, n - 1)$ have been covered within the first n^2 rows of the covering array, but no others. We need to investigate additional stages.

Determine the lowest value of r such that $i \bmod t_r\, g_r \neq j \bmod t_r\, g_r$. This must be true for some value of $r < s$, since $i, j < \prod_{b=0}^{s-1} g_b$, $t_r = \prod_{b=0}^{r-1} g_b$ (by Lemma 6-3), and $i \neq j$. If $g_r = n$, then the arrays $R(n^2 - n, n, n, t_r)$ are used at stage r, where the column duplication factor, $t_r = \prod_{b=0}^{r-1} g_b$. If $g_r = n + 1$, then the arrays $B(n^2 - 1, n + 1, n, t_r)$ are used at stage r.

Consider the values $i^* = \left\lfloor \frac{i \bmod (t_r g_r)}{t_r} \right\rfloor$ and $j^* = \left\lfloor \frac{j \bmod (t_r g_r)}{t_r} \right\rfloor$. The values will be in the range $0 \leq i^*, j^* < g_r$. The value $t_r\, g_r$ represents the width of the arrays (basic or reduced, depending on the value of g_r) that are used at stage r.

Therefore, finding the value modulus $t_r\, g_r$ represents finding the position within a single copy of the array. Since the columns in the arrays are duplicated t_r times consecutively, the value when divided by t_r is equivalent to the column number of the basic array $B(n^2 - 1, n + 1, n, 1)$ or reduced array $R(n^2 - n, n, n, 1)$. Thus, the value of i^* and j^* represent the corresponding column numbers of columns i and j in one of the arrays $B(n^2 - 1, n + 1, n, 1)$ or $R(n^2 - n, n, n, 1)$. In turn, these arrays are both constructed from the orthogonal array $O(n^2, n + 1, n, 2)$.

The goal is therefore to show that $i^* \neq j^*$. We know that $r > 0$, since if $r = 0$, $i \bmod (t_0\, g_0) = j \bmod (t_0\, g_0)$, $t_0 = 1$, $g_0 = n + 1$, and we would have sub-case 1.1.

Therefore, we have $i \bmod (t_{r-1}\, g_{r-1}) = j \bmod (t_{r-1}\, g_{r-1})$, and $i \bmod (t_r\, g_r) \neq j \bmod (t_r\, g_r)$. Since $i \bmod (t_{r-1}\, g_{r-1}) = j \bmod (t_{r-1}\, g_{r-1})$, and $t_r = t_{r-1} \times g_{r-1}$, we have $i \bmod t_r = j \bmod t_r$. We chose r such that $i \bmod (t_r\, g_r) \neq j \bmod (t_r g_r)$, so by Lemma 6-11, it follows that $j \bmod (t_r g_r) - i \bmod (t_r\, g_r) \geq t_r$. Then:

$$\frac{j \bmod (t_r g_r)}{t_r} - \frac{i \bmod (t_r g_r)}{t_r} \geq 1, \text{ so} \left\lfloor \frac{j \bmod (t_r g_r)}{t_r} \right\rfloor \neq \left\lfloor \frac{i \bmod (t_r g_r)}{t_r} \right\rfloor.$$

Therefore, we have $0 \leq i^*, j^* < g_r$, and also $i^* \neq j^*$.

At stage $r > 0$, we have one of two cases. If $g_r = n$, then the values $i \bmod (t_r\, g_r)$ and $j \bmod (t_r\, g_r)$ represent column indices in one of the $(t_r\, g_r)$ copies of the array $R(n^2 - n, n, n, t_r)$ used at stage r. Therefore, i^* and j^* represent indices in the equivalent array with no duplicated columns $R(n^2 - n, n, n, 1)$. By Lemma 6-2, choosing columns $i^* \neq j^*$ in the array $R(n^2 - n, n, n, 1)$ will cover all ordered pairs except (0, 0), (1, 1), ..., $(n - 1, n - 1)$. However, these cases were already covered within the first n^2 rows of C.

If $g_r = n + 1$, then the values $i \bmod (t_r\, g_r)$ and $j \bmod (t_r\, g_r)$ represent column indices in one of the $(t_r\, g_r)$ copies of the array $B(n^2 - 1, n + 1, n, t_r)$ used at stage r. Therefore, i^* and j^* represent indices in the equivalent array with no duplicated

columns

$B(n^2-1, n+1, n, 1)$. By Lemma 6-1, choosing columns $i^* \neq j^*$ in the array $B(n^2-1, n+1, n, 1)$ will cover all ordered pairs except (0, 0). However, this case was already covered by the first row of C.

Therefore, we have covered all ordered pairs of columns i and j, and we can terminate sub-case 1.2.

We have terminated both sub cases 1.1 and 1.2, so therefore, case 1 has been completely examined. Therefore, for $i, j < \prod_{b=0}^{s-1} g_b$ and $i \neq j$, if columns i, and j are selected from $C(\rho, k, n, 2)$, all pair-wise combinations between those columns are covered.

Case 2: One column is "regular" and one column is "extra".

For this case, we have $i < \prod_{b=0}^{s-1} g_b$ and $j \geq \prod_{b=0}^{s-1} g_b$.

We first show that column i is within the part of the array where the orthogonal array $O(n^2, n+1, n, 2)$ was used in the first stage. The width of the set of orthogonal arrays is $\prod_{b=0}^{s-1} g_b$ by Lemma 6-6. We have $i < \prod_{b=0}^{s-1} g_b$ to be in this case, so therefore column i is within the part of the array where the orthogonal arrays are used in the first stage.

Since we also have $j > \prod_{b=0}^{s-1} g_b$, column j must have been added after the orthogonal arrays of the first stage. The only part of Algorithm 6-3 that allows us to add additional columns at the level of the first row is when the Z arrays are added in a stage $r > 0$. Furthermore, the number of rows in each Z array is $\rho \geq n^2$, because there are n^2 rows in the orthogonal arrays used in the initial stage.

Therefore, within the first n^2 rows of the array, column i consists of entries from an orthogonal array, and column j consists of all zeros. All values from 0 to $n-1$ must appear in an orthogonal array column by construction, and therefore, all combinations of the form $(x, 0)$ have been covered between columns i and j in the first n^2 rows.

Let $j^* = j - \prod_{b=0}^{s-1} g_b$. We have $\sum_{b=0}^{r-1} e_b(s) \leq j^* < \sum_{b=0}^{r} e_b(s)$ for some value of r such that $0 < r < s$. This must be true because the total number of columns in C is $\prod_{b=0}^{s-1} g_b + \sum_{b=0}^{s-1} e_b(s)$. Thus, we have identified that column j was added during stage r.

We now examine the rows of C that are added during stage r. First, $(t_{s-1}\, g_{s-1})$ / $(t_r\, g_r)$ copies of the array $R(n^2 - n, n, n, t_r)$ are concatenated horizontally. We know that it must be a reduced array that is used, because if $e_r(s) \neq 0$, then $g_r = n$. The width of the set of reduced arrays is $\prod_{b=0}^{s-1} g_b$ by Lemma 6-6. Therefore, within the rows of C that are added at stage r, column i must consist of a column of the reduced array $R(n^2 - n, n, n, t_r)$.

Since we also have $j \geq \prod_{b=0}^{s-1} g_b$, column j must have been added after the reduced arrays of stage r. The next arrays to be added are $e_q(s) / (\lambda_{qr}\, g_r)$ copies of the array $R(n^2 - n, n, n, \lambda_{qr})$. Since $g_r = n$, the total width of these arrays is $e_q(s)$. Furthermore, this is repeated for $q = 1, \ldots r - 1$, and $e_0(s) = 0$, so the entire width of the R arrays is $\sum_{b=0}^{r-1} e_b(s)$. But, we have $\sum_{b=0}^{r-1} e_b(s) \leq j^* < \sum_{b=0}^{r} e_b(s)$, so column j is not contained within these R arrays. The next array that is concatenated horizontally is $N(n^2 - n, n, e_r(s))$, and this array has $e_r(s)$ columns. Because

$\sum_{b=0}^{r-1} e_b(s) \leq j^* < \sum_{b=0}^{r} e_b(s)$, we must have that column j is, for the rows added at stage r, comprised of a column of the array $N(n^2 - n, n, e_r(s))$.

Therefore, if we look at the rows added at stage r, we have a column of an R array at column i, and a column of an N array at column j. By Lemma 4-4, the column of the R array will consist of $n - 1$ permutations of 0, ..., $n - 1$; and the column of the N array will be n ones, n twos, and so on up to $n - 1$ by construction. This means that all $(x, 1)$, combinations are covered, all $(x, 2)$ combinations are covered, and so on up to $(x, n - 1)$.

Therefore, we have covered all pair-wise combinations for columns i and j, and can terminate case 2.

Case 3: Both columns are "extra".

We have $i, j \geq \prod_{b=0}^{s-1} g_b$. Let $i^* = i - \prod_{b=0}^{s-1} g_b$ and $j^* = j - \prod_{b=0}^{s-1} g_b$.

Since we have i, $j \geq \prod_{b=0}^{s-1} g_b$, both columns must have been added after the orthogonal arrays of the first stage by Lemma 6-6.

We subdivide this case into two sub cases. Sub-case 3.1 is when the two selected columns are added during different stages, and sub-case 3.2 is when the two selected columns are added during the same stage.

Sub-case 3.1: The two selected columns are added during different stages.

Suppose that column i is added during stage r, and column j is added during stage z, where $0 < r < z < s$. Then we have $\sum_{b=0}^{r-1} e_b(s) \leq i^* < \sum_{b=0}^{r} e_b(s)$, and $\sum_{b=0}^{z-1} e_b(s) \leq j^* < \sum_{b=0}^{z} e_b(s)$.

Since both columns are "extra" columns, they will both contain all zeros for at least the first n^2 rows, using the same argument as for column j in case 2. Therefore, we have the combination (0,0) covered.

Next, we shall examine the rows added during stage r. Since $i, j \geq \prod_{b=0}^{s-1} g_b$, columns i and j must have been added after the reduced arrays of stage r, as they have combined width of $\prod_{b=0}^{s-1} g_b$ by Lemma 6-6. The next arrays to be added are $e_q(s) / (\lambda_{qr}\, g_r)$ copies of the array $R(n^2 - n, n, n, \lambda_{qr})$. Since $g_r = n$, the total width of these arrays is $e_q(s)$. Furthermore, this is repeated for $q = 1, \ldots r - 1$, so the entire width of the R arrays are $\sum_{b=0}^{r-1} e_b(s)$. But, we have $\sum_{b=0}^{r-1} e_b(s) \leq i^* < \sum_{b=0}^{r} e_b(s)$, so column i is not contained within these R arrays. The next array that is concatenated horizontally is $N(n^2 - n, n, e_r(s))$, and this array has $e_r(s)$ columns. Because $\sum_{b=0}^{r-1} e_b(s) \leq i^* < \sum_{b=0}^{r} e_b(s)$, we must have that column i is, for the rows added at stage r, comprised of a column of the array $N(n^2 - n, n, e_r(s))$.

Note that we can repeat the above argument to show that for the rows added at stage z, column j is comprised of a column of the array $N(n^2 - n, n, e_z(s))$.

However, we still need to determine what comprises column j for the rows added at stage r, and what comprises column i for the rows added at stage z.

Since $r < z$, we have $\sum_{b=0}^{r} e_b(s) \leq \sum_{b=0}^{z-1} e_b(s) \leq j^*$. Continuing the argument above, at stage r, Algorithm 6-3 stops adding columns when the total reaches $\sum_{b=0}^{r} e_b(s)$. Since we have $\sum_{b=1}^{z-1} e_b(s) \leq j^*$, column j must be added as part of a Z array in a subsequent loop iteration. When the Z array is added, it has ρ rows, where ρ is

the current number of rows in C. In particular, this includes the rows added at stage r, since $r < z$. Therefore, for the rows that are added at stage r, column j consists entirely of zeros.

Finally, what about column i for the rows added at stage z? Since $r < z$, the width of the reduced arrays $R(n^2 - n, n, n, \lambda_{qz})$ added at stage z is $\sum_{b=0}^{z-1} e_b(s)$. But, we have $\sum_{b=0}^{r} e_b(s) \leq \sum_{b=0}^{z-1} e_b(s)$, so column i is within the range of columns formed by the reduced arrays.

To summarize, for the rows added at stage r, we have that column i is comprised of a column of an N array, and column j consists entirely of zeros from an Z array. For the rows added at stage z, we have that column i is comprised of a column of an R array, and column j is comprised of a column of an N array.

Therefore, for the rows added at stage r, we have a column of the N array in column i and part of a column of an Z array at column j. This covers all of the combinations $(1, 0), \ldots, (n-1, 0)$.

Therefore, at this point, we have covered all combinations of the form $(x, 0)$.

Now, let us look at the rows added at stage z. There will be an R array added for column i and an N array for column j. Both arrays contain $n^2 - n$ rows. By Lemma 4-4, the column of the R array will consist of $n - 1$ permutations of $0, \ldots, n - 1$; and the column of the N array will be n ones, n twos, and so on up to $n - 1$ by construction. This means that all $(x, 1)$, combinations are covered, all $(x, 2)$ combinations are covered, and so on up to $(x, n - 1)$.

Therefore, we have covered all pair-wise combinations between columns i and j, and can terminate sub-case 3.1.

Sub-case 3.2: The two selected columns are added during the same stage.

Let columns i and j both be added at stage r. Then we have $\sum_{b=0}^{r-1} e_b(s) \le i^* < j^* < \sum_{b=0}^{r} e_b(s)$ for some r such that $0 < r < s - 1$. Note that r is strictly less than $s - 1$ because $e_{s-1}(s) = t_{s-1}g_{s-1} / t_{s-1}g_{s-1} = 1$. Therefore, we cannot have $\sum_{b=0}^{s-1} e_b(s) \le i^*, j^* < \sum_{b=0}^{s-1} e_b(s) + 1$, for $i^* \neq j^*$.

Let $i^{**} = i^* - \sum_{b=0}^{r-1} e_b(s)$, and $j^{**} = j^* - \sum_{b=0}^{r-1} e_b(s)$.

If the columns i^* and j^* were added during the same stage $r > 0$, then the columns fall within the arrays $Z(\rho, e_r(s))$ for rows added prior to stage r and $N(n^2 - n, n, e_r(s))$ for the rows added during stage r. The array $Z(\rho, e_r(s))$ covers the combination (0, 0), and the array $N(n^2 - n, n, e_r(s))$ covers the combinations (1, 1), …, $(n - 1, n - 1)$. Therefore, all the combinations of the form (x, x) have been covered.

Check if $i^* \bmod g_{r+1} \neq j^* \bmod g_{r+1}$. If this is true, call this sub-case 3.2.1. Otherwise, if $i^* \bmod g_{r+1} = j^* \bmod g_{r+1}$, call this sub-case 3.2.2.

Sub-case 3.2.1:

We shall examine the rows added during stage $r + 1$. We know that there must be such a stage because $r < s - 1$.

Since $i, j \ge \prod_{b=0}^{s-1} g_b$, columns i and j must have been added after the reduced or basic arrays of stage $r + 1$, as they have combined width of $\prod_{b=0}^{s-1} g_b$ by Lemma 6-6.

The next arrays to be added are $e_q(s) / (\lambda_{q,r+1}\ g_{r+1})$ copies of the array $R(n^2 - n, n, n, \lambda_{q,r+1})$ or $B(n^2 - n, n, n, \lambda_{r,r+1})$, depending on the value of g_{q+1}. However, no matter what the value of g_{r+1}, the total width of these arrays is $e_q(s)$. Furthermore, this is repeated for $q = 1, \dots r$, so the entire width of the R arrays are $\sum_{b=0}^{r} e_b(s)$. We have $i^*, j^* < \sum_{b=0}^{r} e_b(s)$, so this tells us that both columns i and j are, for stage $r + 1$, comprised of columns of either $R(n^2 - n, n, n, \lambda_{q,r+1})$ or $B(n^2 - n, n, n, \lambda_{q,r+1})$.

The interpretation of $i^* \bmod g_{r+1} \neq j^* \bmod g_{r+1}$ is that the two columns correspond to different columns in the array $R(n^2 - n , n, n, 1)$ or $B(n^2 - n, n, n, 1)$ added at stage $r + 1$ (note that $\lambda_{r,r+1} = 1$ by Definition 6-10), namely columns $i^* \bmod g_{r+1}$, and $j^* \bmod g_{r+1}$. By Lemma 6-2, choosing columns $i^* \neq j^*$ in the array $R(n^2 - n, n, n, 1)$ will cover all ordered pairs except $(0, 0), (1, 1), \dots, (n - 1, n - 1)$. However, those cases were already covered by the arrays Z and N. Therefore, all pair-wise combinations between columns i and j are covered within C. We can terminate this branch of our sub cases.

Sub-case 3.2.2

The interpretation of $i^* \bmod g_{r+1} = j^* \bmod g_{r+1}$ is that the two columns correspond to identical columns in the array $R(n^2 - n , n, n, 1)$ added at stage $r + 1$. Although we have already covered the combinations (x, x) at the start of sub-case 3.2, we need to investigate additional rows to cover the remaining combinations.

Determine the lowest value of z such that $i^* \bmod (\lambda_{rz}\ g_z) \neq j^* \bmod (\lambda_{rz}\ g_z)$. This must be true for some value of $z < s$, since $\sum_{b=0}^{r-1} e_b(s) \le i^* < j^* < \sum_{b=0}^{r} e_b(s)$, $e_r(s) = \prod_{b=r}^{s-1} g_b$ (by Lemma 6-4) $\lambda_{rz} = \prod_{b=r+1}^{z-1} g_b$ (by Lemma 6-4), and $i^* \neq j^*$. If $g_z = n$, then the arrays $R(n^2 - n, n, n, \lambda_{rz})$ are used at stage z, where the column duplication

factor, $\lambda_{rz} = \prod_{b=0}^{r-1} g_b$. If $g_r = n + 1$, then the arrays $B(n^2 - 1, n + 1, n, \lambda_{rz})$ are used at stage z.

Consider the values $i^{**} = \left\lfloor \frac{i^* \bmod(\lambda_{rz} g_z)}{\lambda_{rz}} \right\rfloor$ and $j^{**} = \left\lfloor \frac{j^* \bmod(\lambda_{rz} g_z)}{\lambda_{rz}} \right\rfloor$. The values will be in the range $0 \le i^{**}, j^{**} < g_z$. The value $\lambda_{rz}\, g_z$ represents the width of the arrays (basic or reduced, depending on the value of g_z) that are used at stage z. Therefore, finding the value modulus $\lambda_{rz}\, g_z$ represents finding the position within a single copy of the array. Since the columns in the arrays are duplicated λ_{rz} times, the value when divided by λ_{rz} is equivalent to the column number of the basic array $B(n^2 - 1, n + 1, n, 1)$ or reduced array $R(n^2 - n, n, n, 1)$. Thus, the value of i^{**} and j^{**} represent the corresponding column numbers of columns i and j in one of the arrays $B(n^2 - 1, n + 1, n, 1)$ or $R(n^2 - n, n, n, 1)$. In turn, these arrays are both constructed from the orthogonal array $O(n^2, n + 1, n, 2)$.

The goal is therefore to show that $i^{**} \neq j^{**}$. We know that $z > r + 1$, since if $z = r + 1$, $i \bmod (\lambda_{r,r+1}\, g_{r+1}) = j \bmod (\lambda_{r,r+1}\, g_{r+1})$, $\lambda_{r,r+1} = 1$ by Definition 6-10, and we would have sub-case 3.2.1.

Therefore, we have $i^* \bmod (\lambda_{r,z-1}\, g_{z-1}) = j^* \bmod (\lambda_{r,z-1}\, g_{z-1})$, and $i^* \bmod (\lambda_{rz}\, g_z) \neq j^* \bmod (\lambda_{rz}\, g_z)$. Since $i^* \bmod (\lambda_{r,z-1}\, g_{z-1}) = j^* \bmod (\lambda_{r,z-1}\, g_{z-1})$, and $\lambda_{rz} = \lambda_{r,z-1} \times g_{z-1}$, we have $i^* \bmod \lambda_{rz} = j^* \bmod \lambda_{rz}$. We chose z such that $i^* \bmod (\lambda_{rz}\, g_z) \neq j^* \bmod (\lambda_{rz}\, g_z)$, so by Lemma 6-11, it follows that $j^* \bmod (\lambda_{rz}\, g_z) - i^* \bmod (\lambda_{rz}\, g_z) \ge \lambda_{rz}$. Then:

$$\frac{j^* \bmod(\lambda_{rz} g_z)}{\lambda_{rz}} - \frac{i^* \bmod(\lambda_{rz} g_z)}{\lambda_{rz}} \ge 1, \text{ so } \left\lfloor \frac{j^* \bmod(\lambda_{rz} g_z)}{\lambda_{rz}} \right\rfloor \neq \left\lfloor \frac{i^* \bmod(\lambda_{rz} g_z)}{\lambda_{rz}} \right\rfloor.$$

Therefore, we have $0 \le i^{**}, j^{**} < g_z$, and $i^{**} \neq j^{**}$.

At stage z, we have one of two cases. If $g_z = n$, then the values i^* mod $(\lambda_{rz}\ g_z)$ and j^* mod $(\lambda_{rz}\ g_z)$ represent column indices in one of the $(\lambda_{rz}\ g_z)$ copies of the array $R(n^2 - n, n, n, \lambda_{rz})$ used at stage z. Therefore, i^{**} and j^{**} represent indices in the equivalent array with no duplicated columns $R(n^2 - n, n, n, 1)$. By Lemma 6-2, choosing columns $i^{**} \neq j^{**}$ in the array $R(n^2 - n, n, n, 1)$ will cover all ordered pairs except (0, 0), (1, 1), ..., $(n - 1, n - 1)$. However, these cases were already covered by the Z and N arrays.

If $g_z = n + 1$, then the values i^* mod $(\lambda_{rz}\ g_z)$ and j^* mod $(\lambda_{rz}\ g_z)$ represent column indices in one of the $(\lambda_{rz}\ g_z)$ copies of the array $B(n^2 - 1, n + 1, n, \lambda_{rz})$ used at stage z. Therefore, i^{**} and j^{**} represent indices in the equivalent array with no duplicated columns $B(n^2 - 1, n + 1, n, 1)$. By Lemma 6-1, choosing columns $i^{**} \neq j^{**}$ in the array $B(n^2 - 1, n + 1, n, 1)$ will cover all ordered pairs except (0, 0). However, this case was already covered by the first row of C.

Therefore, we have covered all pair-wise combinations, and can terminate sub-case 3.2.2.

At this point, we have terminated all sub cases. Therefore, if we select two columns i and j such that $0 \leq i, j < k$, and $i \neq j$, all pair-wise combinations between those columns are covered. Therefore, $C(\rho, k, n, 2)$ generated by Algorithm 6-3 is indeed a covering array of strength 2. □

6.5 Number of configurations generated by the algorithm

In this section, we investigate the number of configurations produced by the construction of $C(\rho, k, n, 2)$ using Algorithm 6-3, which is reproduced on the next page:

$C \leftarrow \varnothing$
For $r = 0, \ldots, s-1$:
  $C \leftarrow C \blacktriangleleft Z(\rho, e_r(s))$; ρ is current number of rows in C.

$$A \leftarrow \begin{cases} O(n^2, n+1, n, 2), \text{ if } r = 0 \\ R(n^2 - n, n, n, t_r), \text{ if } r > 0,\ g_r = n \\ B(n^2 - 1, n+1, n, t_r), \text{ if } r > 0,\ g_r = n+1 \end{cases}$$

  $S \leftarrow \varnothing$
  $S \leftarrow S \blacktriangleleft A$, repeated $(t_{s-1}\, g_{s-1}) / (t_r\, g_r)$ times
  For $q = 1, \ldots, r-1$:

$$A \leftarrow \begin{cases} R(n^2 - n, n, n, \lambda_{qr}), \text{ if } g_r = n \\ B(n^2 - 1, n+1, n, \lambda_{qr}), \text{ if } g_r = n+1 \end{cases}$$

    $S \leftarrow S \blacktriangleleft A$, repeated $e_q(s) / (\lambda_{qr}\, g_r)$ times
  End for
  $S \leftarrow S \blacktriangleleft N(n^2 - n, n, e_r(s))$
  $C \leftarrow C \blacktriangledown S$
End for
□

Let us examine the number of configurations produced this algorithm. For the first stage, we are using the orthogonal arrays $O(n^2, n+1, n, 2)$, and these contain n^2 rows (configurations). In subsequent stages for r from 1 to $s-1$, if $g_r = n$, then $n^2 - n$ configurations are added, while if $g_r = n+1$, then $n^2 - 1$ configurations are added.

The minimum number of configurations generated is:

$n^2 + (s-1)(n^2 - n)$ (when $g_r = n$ for all $r > 0$)

$= n^2 - n + (s-1)(n^2 - n) + n$

$= s(n^2 - n) + n$

$= \lceil \log_{n+1}(k) \rceil (n^2 - n) + n$ (by Lemma 6-9)

Likewise, the maximum number of configurations would therefore be:

$n^2 + (s-1)(n^2 - 1)$ (when $g_r = n+1$ for all $r \geq 0$)

$= n^2 - 1 + (s-1)(n^2 - 1) + 1$

$= s(n^2 - 1) + 1$

$$= \lceil \log_{n+1}(k) \rceil \, (n^2 - 1) + 1 \qquad \text{(by Lemma 6-9)}$$

Therefore, the number of configurations generated, ρ, is such that

$$\lceil \log_{n+1}(k) \rceil \, (n^2 - n) + n \leq \rho \leq \lceil \log_{n+1}(k) \rceil \, (n^2 - 1) + 1$$

The result is logarithmic in the number of parameters k, and quadratic in the value of n, where n is the next integer $\geq n_0$ (the largest number of values) that is a prime power. The logarithm function grows slowly, and so this is a nice result for the number of parameters.

Figure 6-13 shows the results for the algorithm for various combinations of parameters and values. The numbers of values are in the different lines on the graph, while the x-axis is the number of parameters. If you follow the different numbers of values, you can see the quadratic influence of the number of values, but any specific line grows logarithmically with the number of parameters.

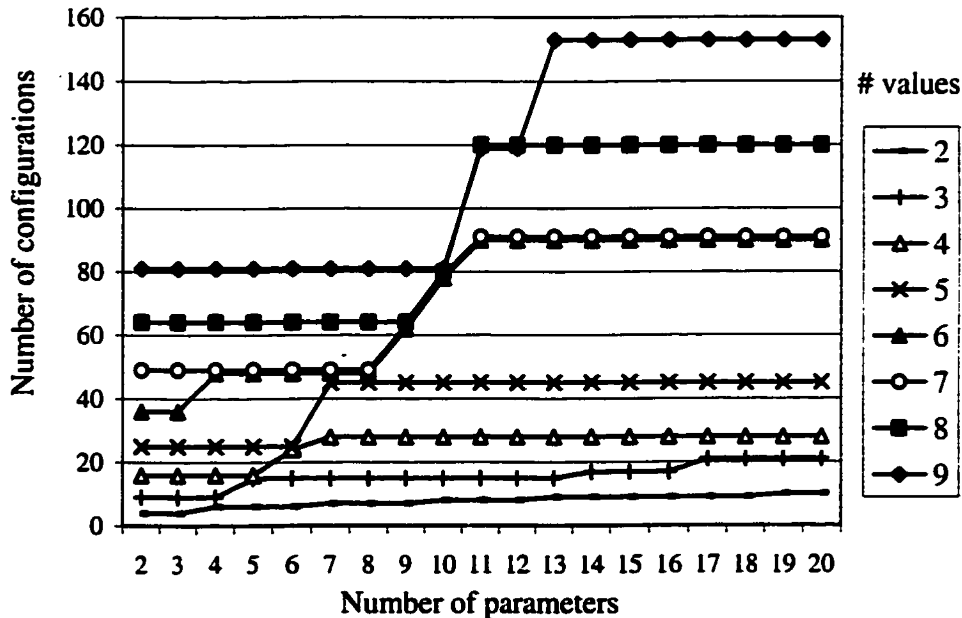


Figure 6-13: Results from the method

6.6 Complexity of the algorithm

In this section, we investigate the algorithmic complexity for generating $C(\rho, k, n, 2)$.

Theorem 6-2 Complexity of constructing $C(\rho, k, n, 2)$

The complexity of constructing $C(\rho, k, n, 2)$ is $O\left(n^3 + \frac{k}{n}(\log_{n+1}(k))^2\right)$.

Proof:

First, we will look at Algorithm 6-3. This algorithm uses the orthogonal, basic, and reduced arrays as building blocks, and we will consider the complexity of the building blocks separately.

Algorithm 6-3 relies on the vertical and horizontal concatenation of arrays as basic operations. We show that with a suitable choice of data structure, we can do the concatenations in linear time in one direction, and in constant time in the other direction. Suppose that each column of an array consists of a linked list containing the column elements, where the list as a whole has a "last item" pointer. Furthermore, the array is then constructed of a linked list of columns, again with a "last item" pointer. A vertical concatenation would consist of joining the linked lists in the corresponding columns. To add elements to the list, we can follow the "last item" pointer for the column in the array on the top, connect the last item to the first list idem in the array on the bottom, and reset the last element pointer. Each column is updated in constant time, and the result is that vertical concatenation is linear in the number of columns. A horizontal concatenation can be done in constant time, since we need only add to the end of the linked list of columns.

Note that we could transpose the array data structure so that vertical concatenation could be done in constant time, and horizontal concatenation could be done in linear time – if this turned out to be to our advantage.

The core of Algorithm 6-3 in terms of significant loops and operations is as follows:

For $r = 0, \ldots, s-1$: executed $\lceil \log_{n+1}(k) \rceil$ times
    $S \leftarrow S \blacktriangleleft A$, repeated $(t_{s-1}\, g_{s-1}) / (t_r\, g_r)$ times
    For $q = 1, \ldots, r-1$:
        $S \leftarrow S \blacktriangleleft A$, repeated $e_q(s) / (\lambda_{qr}\, g_r)$ times
    End for
    $C \leftarrow C \blacktriangledown S$
End for

We can observe that it is advantageous to organize our array data structure so that horizontal concatenation is the faster operation. Therefore, we shall assume that horizontal concatenation can be done in constant time, and vertical concatenation can be done in linear time.

Consider the maximum value of $(t_{s-1}\, g_{s-1}) / (t_r\, g_r)$. The minimum denominator is when $r = 0$, and we have $t_0 = 1$ and $g_0 = n + 1$. Therefore, $(t_{s-1}\, g_{s-1}) / (t_r\, g_r) \leq (t_{s-1}\, g_{s-1}) / (n + 1)$. Now, consider the maximum value of $(t_{s-1}\, g_{s-1})$. From Lemma 6-3, we have $t_{s-1} = \prod_{b=0}^{s-2} g_b$, so $(t_{s-1}\, g_{s-1}) = \prod_{b=0}^{s-1} g_b$. The maximum value of $\prod_{b=0}^{s-1} g_b$ would be $(n + 1)^s$, if $g_b = n + 1$ for all values of b. Now $s = \lceil \log_{n+1}(k) \rceil$, so $s \leq \log_{n+1}(k) + 1$. Therefore, $(n + 1)^s \leq (n+1)^{\log_{n+1}(k)+1}$, which is $(n+1)(n+1)^{\log_{n+1}(k)}$, or $(n + 1)k$. Hence, we have $(t_{s-1}\, g_{s-1}) / (t_r\, g_r) \leq (n + 1)k \,/\, (n + 1)$, and thus $(t_{s-1}\, g_{s-1}) / (t_r\, g_r) \leq k$.

With this result, we have $S \leftarrow S \blacktriangleleft A$, repeated $(t_{s-1}\, g_{s-1}) / (t_r\, g_r)$ times being an operation that takes constant time, repeated on the order of k times.

The innermost loop is a concatenation that is repeated $e_q(s) / (\lambda_{qr}\, g_r)$ times. This quantity would be at its largest when $g_r = n$ for all values of $r > 0$. To minimize the denominator, we can take $\lambda_{qr} = 1$, and $g_r = n$. From Lemma 6-4, $e_q(s) = (t_{s-1}\, g_{s-1}) / (t_r\, g_r)$. We have just shown that $(t_{s-1}\, g_{s-1}) / (t_r\, g_r) \leq k$, so we have $e_q(s) / (\lambda_{qr}\, g_r) \leq k / n$.

Thus, in the innermost loop of Algorithm 6-3, we have a constant-time operation (horizontal concatenation) repeated on the order of k / n times.

At this point, we have the core of Algorithm 6-3 as follows:

For $r = 0, \ldots, s-1$: executed O($\log_{n+1}(k)$) times
  $S \leftarrow S \blacktriangleleft A$, repeated O($k$) times
  For $q = 1, \ldots, r-1$: executed O($\log_{n+1}(k)$) times
    $S \leftarrow S \blacktriangleleft A$, repeated O($k / n$) times
  End for
  $C \leftarrow C \blacktriangledown S$
End for

The inner loop is therefore executed $O\left(\frac{k}{n}(\log_{n+1}(k))^2\right)$ times. In the next loop out, we have two operations that are O(k): the horizontal concatenation, which is a constant time operations repeated up to k times, and the vertical concatenation which is a linear (in k, since this is proportional to the maximum width of C) operation. These take $k\log_{n+1}(k)$ operations, but this term is subordinate to the $O\left(\frac{k}{n}(\log_{n+1}(k))^2\right)$ operations of the inner loop.

Finally, we need to take into account the construction of the original orthogonal arrays that are used as building blocks. Since the orthogonal arrays are based on a set of $n-1$ orthogonal Latin squares of order n, the construction is of order $O(n^3)$ once a Galois field is obtained.

Constructing a Galois field is $O(n)$ in the case where n is a prime number (the elements are integers modulo n). Also, multiplication and addition of two field elements is $O(1)$.

In the case where n is a prime power $n = p^m$ (see Algorithm 4-1), we first need to find a polynomial $q(x)$ of degree m, irreducible in $Z_p[x]$. This can be done by evaluating each possible polynomial, where there are p possible coefficients for m terms. Constructing each of the possible polynomial is $O(p^m) = O(n)$, and evaluating each polynomial takes $O(m)$. Together, this is $O(n^2)$. Once this is done, calculating the field elements is also $O(n^2)$. Producing a multiplication table is $O(n^3)$. This can be referenced when doing the multiplications for calculating each Latin square element. In the worst case, the Galois field

construction will be no worse than $O(n^3)$, and this is done once only before constructing the orthogonal Latin squares.

Therefore, the entire construction of $C(\rho, k, n, 2)$ is of order $O\left(n^3 + \frac{k}{n}(\log_{n+1}(k))^2\right)$. □

6.7 Chapter summary

In this chapter, we have described a recursive approach to the construction of covering arrays.

This chapter has presented an improved algorithm for generating covering arrays of strength two. These covering arrays are constructed to ensure that pair-wise coverage of software components is achieved. The construction has been intuitively illustrated, and then a proof that the construction results in pair-wise coverage has been given.

We have also derived the number of configurations ρ produced by the algorithm, and shown it to be in the range $\lceil \log_{n+1}(k) \rceil\ (n^2 - n) + n \leq \rho \leq \lceil \log_{n+1}(k) \rceil\ (n^2 - 1) + 1$.

We have analyzed the algorithm's complexity, and shown it to be $O\left(n^3 + \frac{k}{n}(\log_{n+1}(k))^2\right)$.

In the next chapter, we investigate adaptations to this construction that are needed to handle situations that arise in practice.

Chapter 7 Comparison among automated methods

7.1 Chapter Introduction

The algorithms from Chapter 6 have been implemented in a tool called "TConfig" (versions exist in C++ and Java). The algorithms are strictly deterministic, and are very fast in practice. This chapter provides results of comparisons with a heuristic method, the "in-parameter order" (IPO) method [Le98], and indirectly with the Telcordia proprietary tool AETG (Automatic Efficient Test Generator) [Coh94a] [Coh94b] [Coh96] [Coh97]. We also investigate the quality of a linear programming approximation to the constraint-based integer programming approach described in section 3.4.2.

TConfig calculates a set of test configurations that will achieve c_2 interaction coverage. The number of configurations produced by this algorithm (see section 6.5) is in the following range, where n is the next integer greater than or equal to n_0 that is a prime power.

Minimum: $\lceil \log_{n+1}(p) \rceil (n^2 - n) + n$

Maximum: $\lceil \log_{n+1}(p) \rceil (n^2 - 1) + 1$

In general, the number of configurations required to achieve c_2 coverage is proportional to the square of the number of values, and the logarithm of the number of parameters.

Before getting to the comparison between TConfig and the IPO method, we first describe the latter in some detail.

7.2 The In-Parameter Order (IPO) method

The "in-parameter order" (IPO) method was developed by Lei and Tai [Le98]. The method was inspired by the well-known greedy heuristic [Jo74] [Chv79] for set covers, but is adapted to the interaction test coverage problem.

The greedy heuristic for set covers is very simple: add the subset that covers the most elements of the set to be covered. Adaptations to this heuristic include checking subsets in decreasing order of cardinality, although this does nothing useful for the interaction coverage problem because all of the subsets are of the same size.

As applied to the interaction coverage problem, this would involve checking each test configuration to see which one covers the most interaction elements. Unfortunately, it is the number of test configurations that increases exponentially, so the direct application of this heuristic is not particularly useful.

Instead, Lei and Tai apply it to choosing parameter values individually, instead of selecting an entire test configuration all at once. They start with a small problem of selecting the first two parameters, and creating $n_0 \times n_1$ test configurations to cover all the combinations among those parameters. (In their case, they do not order the parameters by the number of values; that is, there is no assumption that $n_0 \geq n_1$). They then consider expanding their solution in two dimensions. The first dimension is horizontal. They add another parameter, and then must select new values for the added parameter. These parameters are selected to cover the most number of pairs of parameter values. This phase is referred to as "horizontal growth." After the horizontal growth phase, there may still be uncovered pair-wise combinations between the added parameter and the previous parameters. New test configurations are added to cover the missing combinations, and again a greedy heuristic is used to cover the most uncovered combinations. This phase is called "vertical growth."

The result is a heuristic that is locally greedy in the horizontal and vertical dimensions. Lei and Tai note that the vertical growth is locally optimal, but a fully optimal horizontal growth would be exponential in complexity. This is because entire test configurations would have to be considered, instead of just adding a single parameter value at a time.

Here is a summary of the heuristic:

Algorithm 7-1 The In-Parameter Order (IPO) Algorithm

Main loop:

Start with test set T as the set of interaction elements for the first two parameters.

If there are more than two parameters:

For each additional parameter i:

(T,π) = doHorizontalGrowth(T, i), where π is the set of uncovered interaction elements after this function.

T = doVerticalGrowth(T, π).

Horizontal growth (that is, adding a new parameter i):

Let π = the set of interaction elements associated with the new parameter.

Add a column to T, and set all the values to "-".

For each row up to the minimum of $n_i - 1$, and the number of rows already in T:

Extend the jth test in T by setting $v_i = j$.

$\pi = \pi -$ { interaction elements covered by the extended test }.

For each remaining row in T:

Let $v' = 0$, where v' will save the value that covers the most previously uncovered interaction elements.

Let π' = { interaction elements in π covered by setting the new parameter value to 0 in the jth test in T }.

For each value v for the new parameter, from 1 to $n_i - 1$, do:
Let π'' = { interaction elements in π covered by setting the new parameter value to v in the jth test in T }

If the number of interaction elements in π'' is greater than the previously found maximum, π':
Let $\pi' = \pi''$, and save the new "best" value $v' = v$.

Extend the jth test in T by adding value v'.
$\pi = \pi - \pi'$.

Return T and π.

Vertical growth (adding new tests to cover interaction elements in π):

Let the new set of added tests be $T' = \varnothing$.

For each interaction element { v_j, v_i } in π do (i is the current parameter, j is one of the parameters added previously):

If T or T' contains a test τ with "–" in column i, and (v_j or "–") in column j:

Modify τ by adding v_i, and, if necessary, inserting v_j in column j.

Else

Add a new test to T' that contains the interaction element { v_j, v_i }, and has "–" for every other parameter value.

Add rows T' to T.
Return T.

The complexity of the IPO algorithm is $n^5 \times k^3$ [Le98]. As shown in section 6.6, the complexity of Algorithm 6-3 is $n^3 + \frac{k}{n}(\log_{n+1}(k))^2$, which is a considerable improvement.

7.3 Comparison among methods

The TConfig tool was run using the situations described in the paper by Lei and Tai [Le98]. The results for TConfig have been added to the comparison of the AETG tool [Coh94a] [Coh94b] [Coh96] [Coh97] with the Lei and Tai "Pair-Test" tool in [Le98]. The results are summarized and shown in Table 7-1 and Table 7-2 (which have appeared in [Wi00]).

A distinction is made in this section between the results described in [Le98] of running the "Pair-test" tool, and our own implementation of the IPO method based on the method described in the paper, referred to here as "IPO". Both sets of results are included in Table 7-1 and Table 7-2. To produce a fair comparison, the two programs, TConfig and IPO, were set up to share as many classes as possible in a common class library. The programs were compiled using the same compiler, and run on the same machine – a 333 MHz laptop computer not connected to any network. Since the original Pair-Test results were run on a slower computer, it is to be expected that IPO would run faster on a newer

machine. Our IPO implementation produces slightly different numbers of configurations as well. While there are several cases where IPO produced one extra configuration, there are more cases where the IPO program produced fewer configurations than Pair-Test. The resulting comparison between TConfig and IPO are under as identical conditions as can be expected.

Case #	System	AETG	Pair-Test		IPO		TConfig	
		# config	# config	Time (s)	# config	Time (s)	# config	Time (s)
1	4 parameters, 3 values each	9	9	0.39	10	0.06	9	0.05
2	13 parameters, 3 values each	15	19	0.61	20	0.44	15	0.05
3	61 parameters: 15 with 4 values, 17 with 3 values, 29 with 2 values	41	36	7.53	34	2.75	40	0.05
4	75 parameters: 1 with 4 values, 39 with 3 values, 35 with 2 values	28	29	9.72	27	3.90	30	0.05
5	100 parameters, 2 values each	10	15	11.21	15	3.52	14	0.05
6	20 parameters, 10 values each	180	218	33.89	219	47.46	231	0.44

Table 7-1: Comparison of AETG, Pair-Test / IPO, and TConfig

This set of situations seems to be appropriate for highlighting the differences in the approaches taken by the different tools.

Except for the case 6, variations in the execution time for TConfig were essentially not measurable. However, there is a noticeable increase in the execution time when the number of values was increased substantially – although TConfig was still far faster than Pair-Test or IPO.

TConfig outperforms Pair-Test in three out of six cases, and they produce similar results for a fourth case. Pair-test outperforms TConfig in two cases. AETG produces the fewest configurations in three out of six cases, but is the worst in case 3.

Case 1 represents a standard well-known orthogonal array, which TConfig calculates directly. Pair-Test finds the same result with a heuristic.

Case 2 highlights the situation of covering arrays versus orthogonal arrays. In the first AETG study [Coh94a], the results from this case were reported as 19, but this was upgraded to 15 in [Coh97]. Based on the description in the latter paper, it appears that the AETG heuristics were upgraded in the interval. This case is especially suited to TConfig, since it achieves the result using the building blocks with asymmetric columns.

Case 3 produced fewer configurations than AETG. However, Pair-Test produced the fewest configurations of the three tools.

In case 4, TConfig produced the most configurations, although all three tools are very close. In this case, the TConfig tool used a method from section 8.5, where a covering array was generated assuming that there were 3 values per parameter, and then the fourth value for the largest parameter was added in three extra configurations. Because TConfig uses the building block scheme, it is not particularly effective for cases where there are both a large number of parameters and differing numbers of values for those parameters, and this shows up both in case 3 and case 4. This is an area where further work is needed.

Case 5 highlights the case where there are two values for each parameter. Presumably AETG has implemented an algorithm for the results of Kleitman and Spencer [Kl73] and Katona [Kat75] for $n = 2$, as these papers are cited in the fourth AETG paper [Coh97].

Case 6 exposes a weakness of TConfig, as the number of parameters is just large enough to require a second stage of building blocks. In fact, there could be over 100 parameters with 10 values each, and TConfig would still produce 231 configurations. This case is a candidate for further work (see section 9.3.2).

The cases in Table 7-2 are from a second table in [Le98], which shows only Pair-Test results and times. The TConfig times and results are compared with those from Pair-Test.

Case #	System	Pair-test		IPO		TConfig	
		# Config	time (s)	# Config	time (s)	# Config	time (s)
7	10 parameters, 4 values	31	0.66	31	0.11	28	0.05
8	20 parameters, 4 values	40	1.43	34	0.99	28	0.05
9	30 parameters, 4 values	46	3.46	41	2.42	40	0.05
10	40 parameters, 4 values	49	7.36	42	5.00	40	0.05
11	50 parameters, 4 values	52	14.28	47	9.51	40	0.05
12	60 parameters, 4 values	57	19.55	47	15.98	40	0.05
13	70 parameters, 4 values	60	29.33	49	25.43	40	0.05
14	80 parameters, 4 values	62	40.87.	49	38.50	40	0.05
15	90 parameters, 4 values	62	59.32	52	53.72	43	0.06
16	100 params., 4 values	66	81.07	52	75.02	43	0.11

Table 7-2: Comparison of Pair-Test and TConfig for time and parameter growth

This comparison clearly shows the advantage of TConfig's deterministic algorithm in terms of performance as measured by execution time. Furthermore, the growth of TConfig's execution time is barely measurable over the range of the number of parameters in the comparison. TConfig works especially well for large numbers of parameters, each of which has the same number of values. TConfig will produce, at most, $\lceil \log_{n+1}(k) \rceil (n^2 - 1) + 1$ configurations, so the as the number of parameters grows, the number of test configurations grows only logarithmically. Furthermore, the growth can be predicted with a deterministic algorithm.

Overall, TConfig clearly outperforms the heuristic-based Pair-Test. Against AETG, TConfig produces results that are comparable, except for cases 5 and 6. The difference for case 5 could be eliminated if a special case algorithm was added to TConfig. Case 6 shows a situation where further work is needed. While timed results are not available for AETG, it is unlikely that results are determined as fast as for TConfig, as AETG

examines a significant number of candidate test cases (the number 50 is noted in [Coh97]) before selecting each new test case.

These are clearly very favourable results for TConfig, as it uses the only deterministic algorithm among those compared. TConfig produces predictable and repeatable results, and the number of configurations is generally better than the IPO heuristic, and compares favourably with AETG. Unlike AETG, the algorithms presented here have been published [Wi00][Le98] and are available to be used for further research.

7.4 Approximation of the integer programming problem by linear programming

In section 3.4.2, the interaction test coverage problem was formulated as a $\{0,1\}$ integer programming problem.

To recap the formulation:

- Let $z = | T |$, the number of test configurations.
- Let $y = | X_d |$, the number of interaction elements of degree d.
- Set up variables $\{x_0, \ldots, x_{z-1}\}$, such that the value of $x_j = 1$ if configuration number j is selected, and 0 if configuration number j is not selected.
- Define an array of coefficients A such that $a_{ij} = 1$ if interaction element number i is covered by configuration number j, and 0 otherwise.
- The constraints are $\sum_{j=0}^{z-1} a_{ij} x_j \geq 1$, where $0 \leq i < y$.
- Minimize the objective function $\sum_{j=0}^{z-1} x_j$.

In section 3.4.2, we observed that when the variables x_i are restricted to being integers, the integer programming problem is NP-complete, and only small problems could be solved completely.

The next question that arises is whether there is a useful approximate solution to the integer program. One approach is to relax the constraint that all variables must be integers [Ho82]. This would result in solving a linear program instead of a {0,1} integer program.

We noted in section 3.4.2 that a fractional value for a variable would represent selecting "part" of a configuration. For an exact solution, this is true. However, if we are searching for an approximate solution, we can find the solution to a linear program and round all the fractional values up to one. The rounding process makes this an approximation and not an exact solution.

By a similar argument to the one used in section 3.4.2 for restricting the variables to 0 or 1, the values of variables in the minimized solution of a linear program will be no larger than 1. Instead, they will be in the range $0 \le x_j \le 1$.

A major reason for doing this is that linear programs are much easier to solve in practice than integer programs. We will present some results with the lp_solve [Ber] program (see section 3.4.2 for more details on this program) that support this statement.

In the integer programming results shown in Table 3-2, the case with five parameters and three values for each was killed after running for about six and a half hours. The same input file was run again, but this time the restriction that all 243 variables had to be integers was removed. The resulting linear program was solved in only 0.04 seconds by lp_solve, showing that the problem is much harder when integer solutions are required.

The next step is to discard all the variables that had a value of zero in the solution, and sort the variables in decreasing order of value. This produced the following results:

```
Value of objective function: 9
x043                    0.530351
x047                    0.401344
x117                    0.389934
x201                    0.323851
x069                    0.302597
x241                    0.301478
x213                    0.286215
x003                    0.279507
x176                    0.258343
```

x224	0.25652
x181	0.249479
x107	0.248555
x159	0.247746
x166	0.247169
x130	0.244542
x233	0.243054
x056	0.238081
x113	0.235705
x136	0.234124
x092	0.228301
x099	0.226595
x009	0.173559
x026	0.172155
x013	0.167052
x194	0.165232
x145	0.155093
x058	0.153223
x149	0.152299
x228	0.146957
x156	0.135725
x190	0.129007
x081	0.123935
x100	0.123432
x024	0.120402
x094	0.119709
x119	0.118003
x165	0.11396
x077	0.101096
x005	0.087325
x189	0.0859768
x073	0.0810688
x142	0.0750125
x031	0.0683057
x215	0.0678995
x088	0.0480873
x169	0.0450714
x222	0.04304
x199	0.0277951
x116	0.0118162
x236	0.00895104
x083	0.00532082

Of the 243 original variables, only 51 of them had non-zero values in the linear programming solution. The value of the objective function is 9, because there are nine pair-wise combinations between two parameters that have three values each. However, we can no longer interpret the value 9 as the number of configurations that are required because of the fractional values of the variables.

One approach to take at this point is to take all of the non-zero variables, and give them the value 1. The fractional variable values in the constraints sum to at least one, and so

increasing all of the variable values to one will still satisfy the constraints. Therefore, all interaction elements would be covered.

Intuitively, the program solution tells us that the configurations are only "partly" needed. The result is 51 configurations, which is not a particularly good approximation. However, we have already managed to eliminate about 79% of the configurations from consideration, which is significant.

The values of the variables in the solution of the linear program range from approximately 0.53 (for x_{43}) down to less than 0.006 (for x_{83}). If we start rounding up some of the variables with higher values, the contributions to an integer approximation of the variables with small fractional values may become insignificant. It seems reasonable that in searching for an integer approximation, configuration x_{43} is far more likely to be included than configuration x_{83}.

Therefore, we can try the following approximation strategy: select test configurations in decreasing order of the linear programming solution. After selecting the first nine configurations from the list (at least nine are required for degree 2 interaction coverage), start checking the coverage after each configuration is added. At worst, we are going to wind up with the 51 configurations if they really are all required. However, with some values less than 0.006, this would seem to be unlikely. By selecting the configurations in order, we are keeping the amount that is being added to the objective function (which will eventually represent the number of configurations) to a minimum at each step.

It turns out that after selecting the first 20 configurations from the list $\{x_{43}, x_{47} \ldots x_{92}\}$, degree 2 interaction coverage is achieved. Furthermore, the 16th and 18th configurations on the list (x_{233} and x_{113}) do not add any additional coverage beyond what had been achieved to that point. Therefore, they can be dropped. The result is a set of 18 configurations: $\{x_{43}, x_{47}, x_{117}, x_{201}, x_{69}, x_{241}, x_{213}, x_{3}, x_{176}, x_{224}, x_{181}, x_{107}, x_{159}, x_{166}, x_{130}, x_{56}, x_{136}, x_{92}\}$. Figure 7-1 shows the configurations in the form of parameter values (configuration x_j corresponds to the configuration which is the base 3 representation of $j - 1$):

0	1	1	2	0
0	1	2	0	1
1	1	0	2	2
2	1	1	0	2
0	2	1	1	2
2	2	2	2	0
2	1	2	1	2
0	0	0	0	2
2	0	1	1	1
2	2	0	2	1
2	0	2	0	0
1	0	2	2	1
1	0	2	1	2
2	0	0	1	0
1	1	2	1	0
0	2	0	0	1
1	2	0	0	0
1	0	1	0	1

Figure 7-1: Results from the linear program approximation

Comments on the quality of this approximation will be provided in section 7.5. Theoretical bounds are shown in [Pa91] and [Fe98]. Variations have been suggested in [Sr95] and [Duh97], where instead of selecting the variables in decreasing order of fractional value, the values are selected at random, using the linear programming solution values as probability weights.

The linear programming solver can handle larger problems when the variables are not restricted to integers. However, we are still faced with the exponential growth in the number of variables. Some experiments were run to find out how large a problem could be solved with the tool at hand.

It turns out that the problem with eight parameters and three values could be approximated in this manner, where the problem had $3^8 = 6561$ variables. However, the

problem with 13 parameters and 2 values for each, which has $2^{13} = 8192$ variables, resulted in an "out of memory" indication. This was on a multi-user Unix workstation; memory available obviously depends on what other users were doing at the time. However, for the case with eight parameters and three values, the input to lp_solve is 19,055 lines (72 characters per line) long. Despite the use of sparse-matrix storage, significant memory is still required, as there are 183,708 non-zero values of a_{ij}. This is a property of the problem formulation, and therefore remains an issue for any method to solve the constraints.

Therefore, while the size of the problem that can be handled has increased, it is still true that only the smallest of problems in the general sense can be solved.

7.5 Comparison with the Linear Programming approximation

This section provides results of several experiments comparing like cases for TConfig, our implementation of the IPO method, and the linear programming approximation. Since the linear programming approximation used tools in a different run-time environment than the previous comparisons, execution times are not given. The comparison provides an indication of the quality of the approximation. The results are shown in Table 7-3:

# parameters	# values	TConfig	IPO	Linear Programming approximation
5	3	15	14	18
8	3	15	17	22
10	2	8	9	13

Table 7-3: Comparison among TConfig, IPO, and the linear programming approximation

The results indicate that the linear programming approximation does not result in fewer configurations than either TConfig method or the IPO heuristic. As noted in section 7.4, the linear programming approximation cannot handle problems where there are more than approximately 7,000 possible configurations.

In particular, the case with five parameters and three values for each was chosen, as it is a particularly bad case for TConfig. When the fifth parameter is added, TConfig has to add

six more configurations because of the step function in the number of configurations that are generated. The linear programming approximation still produced more configurations, although IPO managed to produce one less configuration.

7.6 Chapter summary

In this chapter, we have compared Algorithm 6-3 as implemented in the tool TConfig with several other methods for which tools have been developed.

The TConfig tool was compared with the heuristic-based IPO method for the construction of covering arrays. In all cases, the TConfig tool is much faster than the IPO heuristic. For the number of configurations generated, TConfig compares favourably with the IPO heuristic, except for situations where the number of values for each parameter differs considerably. In this situation, it may prove fruitful to investigate a hybrid approach where covering arrays are used to find an initial array based on the smallest number of parameters for each value, and then the IPO method is used to extend the covering array for the extra values for the irregular parameters.

In comparison with the linear programming approximation, TConfig always produced a covering array with fewer configurations. As well, while the size of the problems that could be handled by lp_solve is larger than in the {0,1} integer program situation, there is still the exponential growth of variables in this model of the problem. A commercial tool could handle more variables, but with the exponential growth, the size of problems would still be limited. More importantly, the linear programming approach does not appear to provide a particularly good solution in terms of the number of configurations.

In the next chapter, we turn to practical considerations in the application of these methods.

Chapter 8 Adaptation for practical use

8.1 Chapter Introduction

Much of the work discussed to this point makes several assumptions that may not be satisfied in practice. In this chapter, we consider strategies for adapting covering arrays to fit realistic situations.

To this point, we have assumed that

- Every parameter is independent.
- There are no configurations that are "required" to be present, for various reasons.

The covering array construction in Algorithm 6-3 also makes additional assumptions:

- All parameters have the same number of values.
- The number of values is adjusted to be a prime power, because of the restrictions on the existence of Galois fields used to generate orthogonal or covering arrays.

In practice, none of these assumptions may hold. Therefore, we need strategies to deal with each of these cases.

Another interesting case is when the number of parameters is just slightly in excess of the parameter capacity for using a single-stage orthogonal array. We discuss a strategy that can reduce the number of configurations in this case.

8.2 The number of values is not a prime power

Algorithm 4-4, used to generate the initial orthogonal arrays, depends on the existence of a Galois field of order n. Such fields only exist when n is prime, or a prime power. It is for this reason that we have defined the function $n = f(n_0, \ldots, n_{k-1})$ in Definition 3-3 to determine the number of entries in the covering array, instead of directly using the largest number of values n_0. Our approach to cope with the situation where the largest number of

values for a parameter is neither a prime nor a prime power has been to use the next largest integer that has this property.

An alternative approach could be to use orthogonal arrays with three parameters for cases where n_0 is problematic. One can always construct an orthogonal array with three parameters, since it can be derived from a single Latin square. Therefore, in the case where $n = 6$ (or any other value that is not a prime or prime power), the array $O(n^2, 3, n, 2)$ can be used. (Note that orthogonal arrays with four columns exist for $n > 6$, and n not a prime power [Hal86], but there is no general construction algorithm for them.)

However, this has the effect of restricting the values of g_r to be 2 or 3, and as a result, the parameter capacity grows only by a factor of 2 or 3 for each stage instead of n. This also means that the number of stages, s grows more quickly, as the value changes from $s = \lceil \log_{n+1}(k) \rceil$ to $s = \lceil \log_3(k) \rceil$. The reduced base of the logarithm causes the increase in the number of stages.

This approach results in faster growth of the number of configurations. For example, if we had five parameters and six values for each parameter, we could construct the orthogonal array $O(36, 3, 6, 2)$, and then concatenate two of these arrays horizontally. Next, we can construct the reduced array $R(30, 2, 6, 1)$ by removing the first six rows and the first column of $O(36, 3, 6, 2)$. The array $R(30, 2, 6, 3)$ could then be concatenated vertically in a second stage. This results in 66 configurations.

Instead, if we increase our value of n to seven, we can use five of the columns of the orthogonal array $O(49, 8, 7, 2)$ and a second stage is not required.

The set of prime powers up to 101 is { 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17, 19, 23, 25, 29, 31, 32, 37, 41, 43, 47, 49, 53, 59, 61, 64, 67, 71, 73, 79, 81, 83, 89, 97, 101 }. Suppose that b is the distance to the next prime power for any integer n. If we use the orthogonal array with only three parameters, $2n^2 - n$ is the number of configurations produced by a two-stage construction. However, $(n + b)^2$ is the number of configurations produced by

increasing n to the next prime power. It can be verified that $2n^2 - n >> (n + b)^2$ in all cases above where n is not a prime power.

Therefore, it is preferable to increase n to the value of the next prime power. This results in entries in the covering array that are larger than the actual number of values. The excess values can be changed to "don't care" values.

Furthermore, if n has been increased, we can remove b configurations in most situations with a renumbering of the parameter values. Suppose, again that we have five parameters and six values for each parameter, and we use the right-most five columns of $O(49, 8, 7, 2)$ as our covering array. The first seven configurations are shown in the left half of Figure 8-1. However, Lemma 4-1 tells us that the initial row is guaranteed to be all zeros, and if we do not have to use the first column of an orthogonal array, the first n rows will be all zeros, all ones, and so on up to $n - 1$. Therefore, if we change the values 0 through $b - 1$ to be "don't care" values, and the values of b up to $n - 1$ to be 0, ... , $n - b - 1$ (that is, our original number of values), the first b rows will consist entirely of "don't care" values and can be removed.

Before

0	0	0	0	0
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
4	4	4	4	4
5	5	5	5	5
6	6	6	6	6
...	...	...	...	...

After

–	–	–	–	–
0	0	0	0	0
1	1	1	1	1
2	2	2	2	2
3	3	3	3	3
4	4	4	4	4
5	5	5	5	5
	...	...	...	...

Figure 8-1: Elimination of redundant configuration when n is increased from 6 to 7

Therefore, for five parameters with six values for each, the result after renumbering and discarding the redundant row is 48 configurations.

As noted previously (see section 5.3), we can find covering array $C(37,4,6,2)$ that can handle the case for four parameters and six values for each. If the test suite for each

configuration is large, even small improvements in the number of configurations can result in significant savings for testing. Therefore, it is important to reduce the number of configurations as much as possible, and if smaller covering arrays are known, we can take advantage of them. However, the approach described in this section can be used at any time, and it does not rely on a stored catalogue of known covering arrays. Such a catalogue would improve the performance of the algorithm.

8.3 Number of parameters "just slightly too large"

It turns out that we can make use of the strategy described in the previous section even when the value of n is a prime power. This occurs when we have the situation where the number of parameters is just over the number of parameters that can be handled by an orthogonal array.

Suppose that there are ten parameters, and eight values for each parameter. The array $O(64, 9, 8, 2)$ can only handle up to nine parameters in 64 configurations. Using Algorithm 6-3, a second stage would be added at this point, and this would add another 56 configurations, resulting in 120 configurations.

However, it is to our advantage to increase the value of n in this case, because we are just one parameter over the capacity for a single stage. For the example above, this means we could use $O(81, 10, 9, 2)$, which has 81 configurations. As in section 8.2, we can eliminate a configuration for each increase in the number of values, through appropriate renumbering of parameter values. Thus, the case can be handled with just 80 configurations.

In general, when the number of parameters is just slightly higher than the capacity of an orthogonal array, it is useful to check if we can increase the number of values to keep the construction at a single stage, and use fewer configurations.

8.4 Dependent parameters

A common situation in practice is dependencies among two or more parameters. Specific combinations of parameter values may not be allowed, or one parameter value may

require another parameter to have one of a reduced set of values. The existence of a parameter itself may even be dependent on other parameters having certain values.

The following approach to handling dependent parameters is suggested by Taguchi [Ta87]. Suppose, for example, that there are two types of interfaces between a telephone switch and a telephone. Interface E supports phone models A, B, and C; while interface F supports phone models C and D. Suppose also that there are five versions of switch software, independent of the other network elements. Figure 8-2 illustrates this situation.

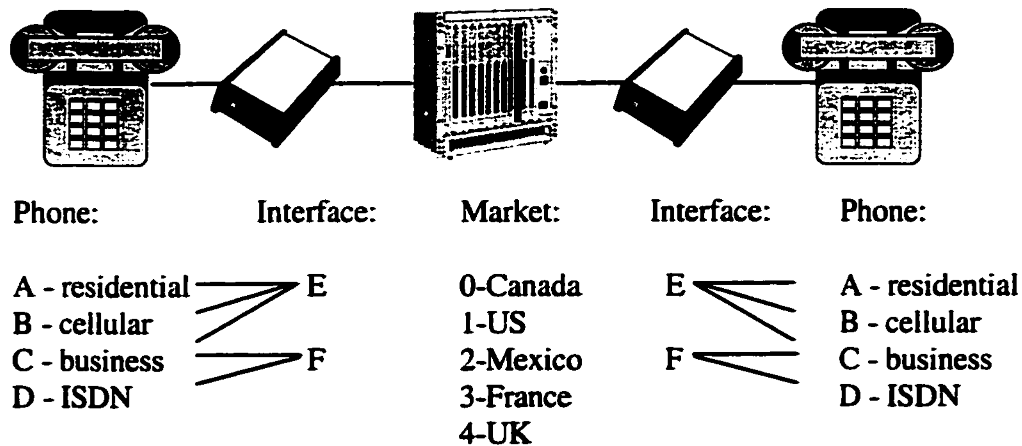


Figure 8-2: Dependent parameters

The objective is to turn the dependent parameters into independent parameters, to be able to apply the theory in the usual manner.

The strategy is to form a hybrid parameter with "values" representing each allowed combination of the dependent parameters. For the interface-phone dependency, construct the renaming in Table 8-1:

Combination	New "value"
$A \times E$	$\Rightarrow 0$
$B \times E$	$\Rightarrow 1$
$C \times E$	$\Rightarrow 2$
$C \times F$	$\Rightarrow 3$
$D \times F$	$\Rightarrow 4$

Table 8-1: Constructing an independent parameter

If values 1 through 5 of the hybrid parameter are covered, then all possible allowed combinations of the interface and telephone dependency are covered. Therefore, the hybrid parameter is independent, and replaces the dependent parameters.

The overall problem therefore reduces to three independent parameters, each of which has five possible values. The covering array $C(25, 5, 3, 2)$ can be applied to this problem. Assuming the switch markets are numbered according to Table 8-1, the configuration (3,2,0) represents the configuration in Table 8-2:

calling phone:	C - business
left interface:	F
switch market:	2 - Mexico
right interface:	E
called phone:	A - residential

Table 8-2: Dependent parameter correspondence

In practice, there may be complex dependencies among parameters and values for which this approach is not useful. This is a problem that merits further study.

8.5 Differing numbers of values

This is another situation that is common in practice.

Orthogonal arrays based on the basic Bose [Bos38] construction of orthogonal Latin squares assume that each parameter has the same number of values, n, and contain n^2 configurations. Suppose instead that for the k parameters, there are n_i possible values of parameter i, which are enumerated as $0, \ldots, n_i - 1$. Without loss of generality, assume that the parameters are ordered so that $n_0 \geq n_1 \geq \ldots \geq n_{k-1}$. Clearly, at least $n_0 \times n_1$ configurations are necessary to achieve pair-wise coverage.

To this point, we have focused on the parameter with the largest number of values. However, we can set the problem up so that the key parameter for selecting the size of the initial orthogonal arrays is the parameter with the **second** largest number of values. That is, we shall base the value of n on n_1 instead of n_0. We can change the definition of the

function $n = f(n_0, \ldots, n_{k-1})$, so that n is the least integer where $n \geq n_1$ and $n = p^m$ for some prime number p and integer $m \geq 1$.

Once we have found a covering array based on n_1, we can add $(n_0 - n) \times n_1$ configurations to complete the array. For example, if we have $k = 4$, $n_0 = 5$ and n_1, n_2, and $n_3 = 3$, we could first construct the array $C(9, 4, 3, 2)$. Then we can add the extra configurations needed to achieve pair-wise coverage, by adding a configuration for each additional value for the over-sized parameter, for each of the values for the other parameters. The result is shown in Figure 8-3.

0	0	0	0	
0	1	1	1	
0	2	2	2	
1	0	1	2	
1	1	2	0	← orthogonal array based on $n_1 = 3$ values
1	2	0	1	
2	0	2	1	
2	1	0	2	
2	2	1	0	
3	0	0	0	
3	1	1	1	
3	2	2	2	← added configurations for values 3 and 4 of n_0
4	0	0	0	
4	1	1	1	
4	2	2	2	

Figure 8-3: Construction based on n_1, with added configurations for n_0

The result is an $n_0 \times n_1$ array that produces the minimal number of test configurations.

If we have the case where there are parameters that have fewer than n_1 values, "don't care" values can be used as required.

However, if there are more than two parameters that have the maximum number of values, and a large number of parameters have fewer than the maximum number of values, additional strategies are needed, and section 9.2 discusses this situation.

8.6 Handling configurations that must be present

In many situations, it is desirable to have certain "required" configurations that must be present in the set of test configurations. There are several reasons for this:

- There is a "recommended" configuration for most users.
- An operational profile has shown that some configurations are commonly used.
- There is already a set of test configurations (for example, for regression testing), and the objective is to add the fewest additional configurations to achieve interaction coverage of a specified degree.

These are comments provided by actual testers during presentations of this material at IBM, and at the Testcom 2000 conference on testing.

An additional advantage of including a "recommended" configuration is that specifications would normally describe the expected system behaviour for this configuration. Therefore, this configuration would be used to debug a test suite.

These are all instances of the same problem: we have to construct a complete covering array, where part of the array has already been specified. However, there is a notable distinction among the above situations. In the first two of the above cases, the number of configurations that are required may be small. In the latter case, there are probably a larger number of required configurations. This difference may affect the strategies that we can apply in these situations.

If the number of required configurations is small, then we may be able to work them into the covering arrays generated by Algorithm 6-3. To this point, we have not discussed any particular strategies for enumerating the actual parameters or values, save for the elimination of dependencies. In the example showing enumeration of parameter values

(Table 4-2), the enumeration is based on nothing but the order of the parameters and values as shown in Figure 4-1.

However, with judicious assignment of parameter numbers and value numbers, we can arrange for a few specified configurations to appear in the set of test configurations. We already know of one configuration that must appear.

Lemma 8-1 **First row of covering array**

Let $C(\rho, k, n, 2)$ be constructed using Algorithm 6-3. Then row 0 of the covering array is all zeros.

Proof:

In Algorithm 6-3, for $r = 0$, the array A used in the horizontal concatenation is the orthogonal array $O(n^2, n + 1, n, 2)$. By Lemma 4-1, row 0 of $O(n^2, n + 1, n, 2)$ will always be all zeros, and so row 0 of array C in Algorithm 6-3 will also be all zeros after the initial stage when $r = 0$. For additional stages when $r > 0$, the array $Z(\rho, e_r(s))$ is horizontally concatenated to C. The array $Z(\rho, e_r(s))$ contains all zeros by construction, and therefore after all stages, the first row of C will contain all zeros. □

Therefore, if there is a particular "recommended" configuration that customers are advised to use, then the particular values for this configuration should each be assigned as value number 0.

This property can be extended in some cases. In [St98b], Stevens includes an extra consideration (which we have not incorporated into the algorithms presented here) for the construction of covering arrays: the largest numbers of *disjoint rows* in the array. A set of rows is disjoint when, if you select any column, no value appears twice in the column. The practical result of having a set of disjoint rows of cardinality x is that the value numbers can be renumbered over the entire covering array so that the first disjoint row is (0 0 … 0), the second disjoint row is (1 1 … 1), and so on up to (x–1 x–1 … x–1). If there are a number of configurations that are required, and have this property of being

disjoint, it may be possible to set up an enumeration of the parameter values to guarantee that the required configurations will appear in the covering array. This will not work if several of the required configurations overlap (that is, a certain parameter value appears in more than one of the required configurations), but for distinct, required configurations, this approach can be applied.

8.7 Inclusion of additional parameters

Another situation that may arise in practice is that additional parameters are added after the initial use of this method to create a set of regression tests. It may be that a new component is added to the system, or some other parameters that were not initially considered are now deemed relevant for setting up test configurations.

This is a situation where using a deterministic approach and the recursive algorithm provide an advantage. The algorithm generates a covering array for the largest possible number of parameters that can be accommodated with a particular set of building blocks. Then only the required number of columns is used.

Therefore, if a new parameter is added, we can usually just take one additional column of the previously generated covering array. Adding this new column does not affect previous columns, and therefore there is no effect on the set of previously existing test configurations.

In practical terms, the modifications to the test suite consist of taking the existing configurations, and selecting a value for the new parameter in each configuration. The new value is assigned according to the added column of the covering array.

If we happen to add a parameter that causes us to exceed the parameter capacity of our previously generated array, we can add another stage to the construction. Because this results only in horizontal and vertical concatenations to the result of the previous stage, there are still no changes to previously assigned configuration values. As before, new values will have to be assigned in the previously existing test configurations. The difference in this case is that new test configurations will be added.

Because the method does not require any modifications to previously assigned test configuration values, the method is particularly suited to this situation.

8.8 Inclusion of additional values

While the inclusion of additional parameters is facilitated by this approach, the method is not as well suited to inclusion of additional values.

If a new value is added to a single parameter, then one can always just add the extra configurations that are necessary to the covering array that had been generated previously. For example, if we originally had four parameters, and three values for each, the original covering array would be $C(9, 4, 3, 2)$, which contains nine configurations. Adding a fourth value 3 to the parameter 2 could be handled just by adding the configurations { 0, 0, 3, 0 }, { 1, 1, 3, 1 }, and { 2, 2, 3, 2 }; that is, adding a configuration with the new parameter value and each of the original values of the other parameters. The resulting set of twelve configurations would be the best that is possible, since at least 4 x 3 = 12 configurations would be required for degree 2 interaction coverage. However, as this process is repeated, the quality of the covering array would deteriorate. If a fourth value was added to another parameter, and we repeated this process, the resulting 16 configurations are still the best that we could do. However, if a third parameter has a fourth value added, then we have reached the point where we depart from the minimum number of configurations. At this point, we know that that the covering array $C(16, 5, 4, 2)$ exists, and we could handle up to five parameters with four values each in just 16 configurations. Unfortunately, the use of this array does not preserve the original set of test configurations when using the algorithms presented here.

This shows that there is a small capacity to add new values to new parameters without the number of configurations being affected adversely (in the sense of regenerating the covering array to obtain fewer configurations). However, this capacity is quickly used up, and at this point, a user would have to evaluate whether the greater cost is entailed with testing more than the necessary number of configurations, or re-generating the entire set of configurations. If there is considerable investment already in a regression test

suite, the better option may be to continue with the incremental adding of configurations as needed.

There are other ways that capacity for additional values may already be available in a set of configurations. The presence of "don't care" values may be a result of having previously increased the value of n. This could be because the largest number of values was not a prime power, or that using a slightly larger orthogonal array would result in fewer configurations than a two-stage construction of a covering array. In these cases, the original values that did not really exist would appear as "don't care" values. However, if new values are then added subsequently, then the "don't care" values could be changed back to the original numbers.

In addition, although Algorithm 6-3 specifies the use of the Z array that is all zeros, it will be observed from Figure 6-10 that some of the zeros in the Z array could actually be "don't care" values. Parameters represented by such columns would then have some additional capacity to add new values.

However, while the above strategies can be employed on an incremental basis, and repeated a limited number of times, the method of covering arrays is not particularly suited to the addition of large numbers of new values to a single parameter, or to adding new values to a large number of parameters. This is an area where more investigation is needed.

8.9 Chapter summary

In this chapter, we have investigated some of the situations that arise in practice where adaptations are needed to the theoretical approach. The methods presented in this chapter can be used to fit the methods to practical situations, and we have not encountered a case where the covering array approach cannot be used. Of course, the quality of the "fit" will vary depending on the situation.

The approaches for handling dependencies among parameters, or additional parameters that are added subsequently, work quite well with the method of covering arrays. In

addition, we have a configuration reduction strategy when the number of parameters is just slightly in excess of the maximum number of parameters that can be handled by a single-stage orthogonal array.

Situations where the practical adaptation is workable, but may not provide the best results, are when additional values are added subsequently, or there are significant differences in the number of values for each parameter. Some of the latter c[illegible] will be discussed in Chapter 9 as future work, as well as a multi-stage strategy for cases when the number of parameters is just slightly over the limit for a particular number of stages.

Chapter 9 Conclusions and suggestions for further research

9.1 Chapter Introduction

In this chapter, we summarize the results of our work, and outline the areas for which further research into algorithms for generating covering arrays is needed. In particular, we mention a few areas where there appears to be a promising approach, and where results in an area seem to be particularly needed.

9.2 Conclusions

In this thesis, we have addressed the problem of testing interactions among components of a software system. A strategy is needed when the number of test configurations is so large that testing of all configurations is not possible within any realistic budget of time or money. The strategy proposed in this work has the potential to save significant resources in testing component interactions, and provides a metric for the measurement of interaction coverage.

The problem studied in this thesis is a general problem: it occurs in any system testing situation where there are a number of parameters that comprise a test configuration, and each parameter has a set of discrete values. Chapter 2 has described a number of specific situations where this arises in practice.

In Chapter 3, a theoretical framework was developed for the interaction test coverage problem, through the introduction of the concept of an interaction element. This concept abstracts the selection of a subset of the parameters, and the selection of specific values for the parameters in the subset. The interaction test coverage problem turns out to be an instance of the minimum set cover problem, and an instance of the $\{0,1\}$ integer programming problem. However, it is not possible to convert these latter two problems, in the general case, to an interaction test coverage problem. The question of whether the interaction test coverage problem is NP-complete remains open at this point in time. The

theoretical framework leads to a metric for measuring test coverage: the percentage of interaction elements that are covered by a set of test configurations.

In Chapter 4, the methods of statistical experimental design were reviewed, and the construct of an orthogonal array was examined. The situations where such arrays have been used traditionally were described. A well-known algorithm for construction of orthogonal arrays was also described.

In Chapter 5, we showed that the constructs of traditional experimental design could be adapted for the purposes of testing interactions among software components. The similarities and differences of the situations were highlighted. In particular, the property that system testing produces a discrete test verdict that is independent of the experimental design – that is, it is determined through reference to system specifications – is key to the modified experimental design proposed in this work. This property allows the use of a covering array instead of an orthogonal array. When there are large numbers of parameters relative to the number of values, a covering array has no more, and often far fewer, rows than an orthogonal array with a sufficient number of columns. Therefore, the use of covering arrays can produce fewer test configurations. There are also fewer restrictions on the existence of covering arrays, and therefore they can be more easily fitted to a particular problem.

In Chapter 6, we developed a fast, deterministic, algorithm to generate a covering array for an arbitrary number of parameters that includes all pair-wise interactions among parameter values. This algorithm is based on using subsections of orthogonal arrays as building blocks for a multi-stage construction algorithm. A proof was given that the algorithm generates an array of the requisite size, and that all pair-wise interactions are indeed covered. The number of configurations ρ produced by the algorithm is shown to be in the range $\lceil \log_{n+1}(k) \rceil\ (n^2 - n) + n \leq \rho \leq \lceil \log_{n+1}(k) \rceil\ (n^2 - 1) + 1$. The complexity of the algorithm is shown to be $O\left(n^3 + \frac{k}{n}(\log_{n+1}(k))^2\right)$.

In Chapter 7, we compared the method proposed herein, with the In-Parameter Order (IPO) heuristic, which also generates a set of configurations that achieves pair-wise coverage. We showed that the deterministic construction of covering arrays is always faster, and, in most cases produces fewer configurations. The method was also compared to a linear programming approximation, and it was shown that the method proposed herein produces fewer configurations, and can handle problems of arbitrary size. The linear programming approximation can handle slightly larger problems than the exact {0,1} integer programming solution, but even the linear programming approximation is still restricted to the smallest of system configurations.

In Chapter 8, we discussed adaptations to the theory that can be used to apply the method in practice. These include the handling of dependencies among parameters, the subsequent addition of new parameters or new values to a regression test suite, the case where the number of values for each parameter is different, the inclusion of "required" configurations, and several situations where the number of configurations could be further reduced.

Overall, the major contributions of this thesis are as follows:

1. A formal definition of the interaction test coverage problem is presented.

2. A metric for measurement of interaction test coverage is defined. The objective is to cover all interactions of a specified degree, and the percentage of such interactions covered forms the metric.

3. The interaction test coverage problem is related to some well-known combinatorial optimization problems, and the NP-completeness of the problem is discussed. While the interaction coverage problem is a special case of the minimum set cover, and {0, 1} integer programming problems, neither of the latter problems can be transformed to the interaction test coverage problem. The question of whether the interaction test coverage problem is NP-complete remains open.

4. The relationship between software system interaction testing and statistical experimental design is developed.

5. A new, fast, deterministic, algorithm is developed for achieving pair-wise interaction coverage, and compared with other methods. Because the method is deterministic, it produces predictable, repeatable results.

6. A number of areas for further research have been identified.

9.3 Suggestions for further work

In this section, we outline a number of areas which merit further investigation to improve on the results presented here.

These areas are:

- Differing numbers of parameter values, especially when there are significant differences and a large number of parameters.
- Large, multi-stage covering arrays.
- Additional initial building blocks.
- Algorithms for covering arrays that have strength greater than two.

9.3.1 Differing numbers of parameter values.

Further work needs to be done for situations where the number of parameter values varies widely, especially when there are large numbers of parameters. In this work, we used the function $f(n_0, \ldots, n_{k-1})$ to be the next integer greater than or equal to n_0 (or n_1) that is a prime power. An algorithm based on the full set of values $n_0, \ldots, n_{k-1}$ should produce better results when these values vary significantly.

This issue needs to be addressed in particular for the recursive construction of large covering arrays. If we have the situation where $n_0 > n_1$ and $n_1 = \ldots = n_{k-1}$ (one parameter with extra values), this specific case is addressed in section 8.5. However, the general case where $n_0 \geq n_1 \geq \ldots \geq n_{k-1}$ needs more investigation. An example of this type of

situation is case 3 in the tool comparison in Table 7-1. At this point, the recursive scheme does not take into account differing numbers of parameters, especially in determining the sizes and numbers of building blocks.

One approach to be investigated when there are a few unbalanced parameters involves the use of a full orthogonal array, and then combining parameters that cover interactions. This can handle only a limited amount of parameters, but could be adapted as another building block for recursive covering array construction.

For example, $O(8,7,2,2)$, shown in Table 9-1, can be converted to handle five parameters: four that have two values, and one that has four values, shown in Table 9-2.

Configuration	Parameters						
Number	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	1
2	0	1	1	0	0	1	1
3	0	1	1	1	1	0	0
4	1	0	1	0	1	0	1
5	1	0	1	1	0	1	0
6	1	1	0	0	1	1	0
7	1	1	0	1	0	0	1

Table 9-1: The orthogonal array $O(8,7,2,2)$

Configuration	Original parameters				
Number	012	3	4	5	6
	Modified Parameters				
	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	1	0	0	1	1
3	1	1	1	0	0
4	2	0	1	0	1
5	2	1	0	1	0
6	3	0	1	1	0
7	3	1	0	0	1

Table 9-2: $O(8,7,2,2)$ modified for one parameter with four values

Two investigations are needed here. The first is whether the method of combining columns in an orthogonal array can be adapted to covering arrays. This would give us a useful method that can be used when there are only a few parameters with larger number of values.

The second investigation is how to handle the general case of differing number of values in the case where there are a large number of parameters. In particular, it may be useful to have building blocks of various sizes to handle parameters of different sizes.

9.3.2 Numbers of parameters "just slightly too large"

This case was discussed previously in sections 8.2 and 8.3, for the situation where the number of parameters was just slightly in excess of the capacity for the initial stage. However, there need to be additional strategies to also cover the case where the number of parameters is just in excess of the parameter capacity of a specific stage $r \geq 1$.

The need for these strategies is illustrated in case 6 from the tool comparison in Table 7-1. In this case, there are 20 parameters and 10 values for each parameter. For 10 values, we can handle up to 12 parameters in 121 configurations ($n = 11$ is used here

since there is no Galois field of size 10). Adding a thirteenth parameter would normally force a second stage. This would add another 110 configurations for a total of 231. If we used the strategy of increasing the number of values, we would have to use the orthogonal array for 20 parameters with 19 values, $O(361, 20, 19, 2)$. However, 361 configurations do not save us anything. The evidence from AETG shows us that no more than 180 configurations are necessary (a saving of 51 configurations, as compared with 231), so clearly there is potential for improvements in these situations.

9.3.3 Issues with block sizes

Some experiments we have run produced results showing that the building blocks chosen by Algorithm 6-2 are not necessarily the best for the situation. There are two situations when this appears to happen. The first is for very large multistage arrays. The second is for a number of parameters that is just over the limit for a particular number of stages, and requires an extra stage to handle only a few parameters.

When there are enough stages in a multistage covering array, it may turn out that it is better to use more stages that add fewer configurations at each stage, rather than using fewer stages. More investigation is needed into how to make use of this property.

Here is an example. Suppose that there are 48 parameters and 2 values for each parameter. One construction that is possible, and would be produced by Algorithm 10, is:

<table>
<tr><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td colspan="3">B(3,3,2,3)</td><td colspan="3">B(3,3,2,3)</td><td colspan="3">B(3,3,2,3)</td><td colspan="3">B(3,3,2,3)</td><td colspan="3">B(3,3,2,3)</td><td>*R(3,3,2,3)</td></tr>
<tr><td colspan="9">B(3,3,2,9)</td><td colspan="7">*B(3,3,2,9)</td></tr>
<tr><td colspan="16">*R(2,2,2,27)</td></tr>
</table>

Figure 9-1: $C(12,48,2,2)$ constructed in four stages

In Figure 9-1, O stands for $O(4, 3, 2, 2)$. Arrays marked with a * are truncated on the right, as not all of their columns are required. Using all of the columns would result in being able to handle 54 parameters. (In fact, the array where R appears in the table could actually be filled with "don't care" values in this particular case.) The objective in

Algorithm 10 is to use the fewest number of stages, and the result is 12 configurations. In the five stages, 4, 3, 3, and 2 configurations are used respectively.

An alternative approach is shown in Figure 9-2. As it turns out, in this case, exactly the same number of configurations is produced, but with one extra stage. Because reduced arrays (which add two test configurations) are used at each stage, the growth is slower than when using basic arrays, which add three configurations at each stage.

<table>
<tr><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td colspan="2">R</td><td colspan="2">R</td><td colspan="2">R</td><td colspan="2">R</td><td colspan="2">R</td><td colspan="2">R</td><td colspan="2">R</td><td colspan="2">R</td></tr>
<tr><td colspan="4">R(2,2,2,6)</td><td colspan="4">R(2,2,2,6)</td><td colspan="4">R(2,2,2,6)</td><td colspan="4">R(2,2,2,6)</td></tr>
<tr><td colspan="8">R(2,2,2,12)</td><td colspan="8">R(2,2,2,12)</td></tr>
<tr><td colspan="16">R(2,2,2,24)</td></tr>
</table>

Figure 9-2: $C(12, 48, 2, 2)$ constructed in five stages

In Figure 9-2, O stands for $O(4, 3, 2, 2)$, and R stands for $R(2, 2, 2, 3)$.

When this trend is extrapolated, it appears that growing in steps of two configurations at a time and using more stages will produce better results than using fewer stages that add three configurations at a time. Investigation is needed as to the relationship between k and n that produces this effect.

In addition, there are occasions where increasing the number of columns in the order of the stages may not produce the best result. For example, suppose we have 52 parameters, each of which have three values. If we apply Algorithm 6-2 to select the building blocks, the result will be the construction in Figure 9-3:

<table>
<tr><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td></tr>
<tr><td colspan="4">B(8, 4, 3 ,4)</td><td colspan="4">B(8, 4, 3 ,4)</td><td colspan="4">B(8, 4, 3 ,4)</td><td>B*</td></tr>
<tr><td colspan="13">B(8, 4, 3, 12)</td></tr>
</table>

Figure 9-3: $C(25, 52, 3, 2)$ constructed using Algorithm 6-2

In Figure 9-3, O stands for $O(9, 4, 3, 2)$, and B^* stands for the first four columns of $B(8, 4, 3, 4)$. This construction is selected because the parameter capacity for using an R array in the third stage is only 49.

However, it is possible to extend the construction algorithm in two ways. The first is to permit the use of basic arrays in stages subsequent to the use of reduced arrays. That is, we would not be increasing the column capacity in the order of the stages. The second extension would be to allow extra columns when a subsequent stage will use basic arrays. This means that we may need a basic array to ensure pair-wise coverage of extra columns, instead of a reduced array.

If we use both of these extensions, we can construct $C(23, 52, 3, 2)$ as shown in Figure 9-4:

<table>
<tr><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>O</td><td>Z(9, 4)</td></tr>
<tr><td colspan="3">R(6, 3, 3, 4)</td><td colspan="3">R(6, 3, 3, 4)</td><td colspan="3">R(6, 3, 3, 4)</td><td colspan="3">R(6, 3, 3, 4)</td><td>N(6, 4, 3, 4)</td></tr>
<tr><td colspan="12">B(8, 4, 3, 12)</td><td>B(8, 4, 3, 4)</td></tr>
</table>

Figure 9-4: $C(23, 52, 3, 2)$ constructed using algorithm extensions

The result is that we have saved two configurations. Further work is needed to detect when this sort of situation exists in the general case, and then to modify the algorithm to include these situations.

9.3.4 Additional initial stage building blocks

Additional work is also needed to investigate how to use other initial building blocks beyond $O(n^2, n + 1, n, 2)$. There is no requirement that orthogonal arrays be used as the initial building blocks, although their use to this point has yielded good results.

For example, because six is not a prime power, there is no $O(36, 7, 6, 2)$. (In fact, $O(36, 4, 6, 2)$ does not exist either.) The algorithm presented here would use $n = 7$ in this case. The trade-off here is that a library of "known" building blocks would need to be kept. The algorithms presented here do not rely on a stored library, or on searching.

Stevens, in [St98b], presents a list of covering arrays that have been found by hand, or a simulated annealing algorithm. These arrays can then be used, as we have used orthogonal arrays here, to construct larger covering arrays.

In particular, the following small covering arrays have been found:

	3 values	4 values	5 values
4 parameters			
5 parameters	$C(11, 5, 3, 2)$		
6 parameters		$C(19, 6, 4, 2)$	
7 parameters	$C(12, 7, 3, 2)$	$C(21, 7, 4, 2)$	$C(29, 7, 5, 2)$
8 parameters		$C(23, 8, 4, 2)$	$C(34, 8, 5, 2)$
9 parameters	$C(13, 9, 3, 2)$	$C(24, 9, 4, 2)$	$C(37, 9, 5, 2)$
10 parameters		$C(25, 10, 4, 2)$	$C(42, 10, 5, 2)$

Table 9-3: Small covering arrays found by simulated annealing

In particular, the simulated annealing algorithm discovered $C(11, 5, 3, 2)$, and $C(12, 7, 3, 2)$, which were not previously known to exist.

Another small covering array is $C(37, 4, 6, 2)$, described in [St98a], and presented here in section 5.3.

However, the simulated annealing algorithm does not appear to be effective for $n > 6$ [St00], and for $n = 7$, the best known results in [St98b] match those that can be constructed by TConfig, using the method of Algorithm 6-3.

Additional work is needed to investigate in which situations these smaller building blocks could be used to construct larger arrays. This would require modifications to Algorithm 6-1 and Algorithm 6-2 that describe how to select which building blocks can be used at each stage. However, the core of Algorithm 6-3 that constructs the large covering array from the selected building blocks should remain essentially the same.

9.3.5 Generalization of construction algorithms to *d*-way interactions

The recursive scheme (Algorithm 6-3) presented in section 6.3 constructs covering arrays $C(\rho, k, n, 2)$. It would be useful to extend this to the general case $C(\rho, k, n, d)$. This would provide the capability of generating a set of test configurations for various degrees of interaction coverage, so that the number of test configurations can be adjusted to cover the maximum degree of interactions within specific budget constraints.

Initially, it appears that the same general approach can be taken, in that if one starts with $O(n^d, n+1, n, d)$, multiple copies of this array could be concatenated horizontally in a first stage. Since all d-way interactions are covered inside the orthogonal array, they should also be covered among like columns in other copies of the array.

However, instead of having missing pair elements, there will be missing d-interaction elements. In section 6.2.2, we used the strategy of covering pair elements by adding a second stage by adding the non-redundant parts of an orthogonal array with the columns duplicated. As an example, we shall try this approach with the case with nine parameters, and two values for each parameter. First, we horizontally concatenate three copies of the orthogonal array $O(8,3,2,3)$. Then we use the same array in a second stage, but duplicate all columns three times. (There will be two rows of all zeros and all ones that result; these redundant rows can be dropped.) The result is shown in the following TConfig output excerpt:

```
Degree of interaction coverage:  3
Number of parameters:            9
Number of values:                2
Number of configurations:        14
```

0	0	0	0	0	0	0	0	0
0	1	1	0	1	1	0	1	1
0	0	1	0	0	1	0	0	1
0	1	0	0	1	0	0	1	0
1	0	1	1	0	1	1	0	1
1	1	0	1	1	0	1	1	0
1	0	0	1	0	0	1	0	0
1	1	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1
0	0	0	0	0	0	1	1	1
0	0	0	1	1	1	0	0	0
1	1	1	0	0	0	1	1	1
1	1	1	1	1	1	0	0	0
1	1	1	0	0	0	0	0	0

```
1:  Missing combination (1, 0, 0) for parameters 0, 1, and 3.
2:  Missing combination (0, 1, 1) for parameters 0, 1, and 3.
3:  Missing combination (0, 1, 0) for parameters 0, 1, and 4.
4:  Missing combination (1, 0, 1) for parameters 0, 1, and 4.
5:  Missing combination (1, 0, 0) for parameters 0, 1, and 6.
6:  Missing combination (0, 1, 1) for parameters 0, 1, and 6.
7:  Missing combination (0, 1, 0) for parameters 0, 1, and 7.
8:  Missing combination (1, 0, 1) for parameters 0, 1, and 7.
9:  Missing combination (1, 0, 0) for parameters 0, 2, and 3.
10:  Missing combination (0, 1, 1) for parameters 0, 2, and 3.
11:  Missing combination (0, 1, 0) for parameters 0, 2, and 5.
```

```
12: Missing combination (1, 0, 1) for parameters 0, 2, and 5.
13: Missing combination (1, 0, 0) for parameters 0, 2, and 6.
14: Missing combination (0, 1, 1) for parameters 0, 2, and 6.
15: Missing combination (0, 1, 0) for parameters 0, 2, and 8.
16: Missing combination (1, 0, 1) for parameters 0, 2, and 8.
17: Missing combination (0, 1, 0) for parameters 0, 3, and 4.
18: Missing combination (1, 0, 1) for parameters 0, 3, and 4.
19: Missing combination (0, 1, 0) for parameters 0, 3, and 5.
20: Missing combination (1, 0, 1) for parameters 0, 3, and 5.
21: Missing combination (0, 1, 0) for parameters 0, 6, and 7.
22: Missing combination (1, 0, 1) for parameters 0, 6, and 7.
23: Missing combination (0, 1, 0) for parameters 0, 6, and 8.
24: Missing combination (1, 0, 1) for parameters 0, 6, and 8.
25: Missing combination (1, 0, 0) for parameters 1, 2, and 4.
26: Missing combination (0, 1, 1) for parameters 1, 2, and 4.
27: Missing combination (0, 1, 0) for parameters 1, 2, and 5.
28: Missing combination (1, 0, 1) for parameters 1, 2, and 5.
29: Missing combination (1, 0, 0) for parameters 1, 2, and 7.
30: Missing combination (0, 1, 1) for parameters 1, 2, and 7.
31: Missing combination (0, 1, 0) for parameters 1, 2, and 8.
32: Missing combination (1, 0, 1) for parameters 1, 2, and 8.
33: Missing combination (1, 1, 0) for parameters 1, 3, and 4.
34: Missing combination (0, 0, 1) for parameters 1, 3, and 4.
35: Missing combination (0, 1, 0) for parameters 1, 4, and 5.
36: Missing combination (1, 0, 1) for parameters 1, 4, and 5.
37: Missing combination (1, 1, 0) for parameters 1, 6, and 7.
38: Missing combination (0, 0, 1) for parameters 1, 6, and 7.
39: Missing combination (0, 1, 0) for parameters 1, 7, and 8.
40: Missing combination (1, 0, 1) for parameters 1, 7, and 8.
41: Missing combination (1, 1, 0) for parameters 2, 3, and 5.
42: Missing combination (0, 0, 1) for parameters 2, 3, and 5.
43: Missing combination (1, 1, 0) for parameters 2, 4, and 5.
44: Missing combination (0, 0, 1) for parameters 2, 4, and 5.
45: Missing combination (1, 1, 0) for parameters 2, 6, and 8.
46: Missing combination (0, 0, 1) for parameters 2, 6, and 8.
47: Missing combination (1, 1, 0) for parameters 2, 7, and 8.
48: Missing combination (0, 0, 1) for parameters 2, 7, and 8.
49: Missing combination (1, 0, 0) for parameters 3, 4, and 6.
50: Missing combination (0, 1, 1) for parameters 3, 4, and 6.
51: Missing combination (0, 1, 0) for parameters 3, 4, and 7.
52: Missing combination (1, 0, 1) for parameters 3, 4, and 7.
53: Missing combination (1, 0, 0) for parameters 3, 5, and 6.
54: Missing combination (0, 1, 1) for parameters 3, 5, and 6.
55: Missing combination (0, 1, 0) for parameters 3, 5, and 8.
56: Missing combination (1, 0, 1) for parameters 3, 5, and 8.
57: Missing combination (0, 1, 0) for parameters 3, 6, and 7.
58: Missing combination (1, 0, 1) for parameters 3, 6, and 7.
59: Missing combination (0, 1, 0) for parameters 3, 6, and 8.
60: Missing combination (1, 0, 1) for parameters 3, 6, and 8.
61: Missing combination (1, 0, 0) for parameters 4, 5, and 7.
62: Missing combination (0, 1, 1) for parameters 4, 5, and 7.
63: Missing combination (0, 1, 0) for parameters 4, 5, and 8.
64: Missing combination (1, 0, 1) for parameters 4, 5, and 8.
65: Missing combination (1, 1, 0) for parameters 4, 6, and 7.
66: Missing combination (0, 0, 1) for parameters 4, 6, and 7.
67: Missing combination (0, 1, 0) for parameters 4, 7, and 8.
68: Missing combination (1, 0, 1) for parameters 4, 7, and 8.
```

```
69:  Missing combination (1, 1, 0) for parameters 5, 6, and 8.
70:  Missing combination (0, 0, 1) for parameters 5, 6, and 8.
71:  Missing combination (1, 1, 0) for parameters 5, 7, and 8.
72:  Missing combination (0, 0, 1) for parameters 5, 7, and 8.
```

We have missed covering 72 interaction elements, as listed above. These interaction elements have the common property that the missing interaction element contains two columns in one of the original orthogonal array copies, and one column from another of the copies. This type of interaction element does not appear to be targeted in our strategy, and a more comprehensive strategy is required to cover the missing interaction elements in multi-stage constructions. An approach such as that described in [Cha99] begins the work in this direction.

9.3.6 The case with 2 values

As noted in section 5.4.1, the case for two values has been solved completely. A full set of algorithms would include an algorithm based on the result in [Sl93]. This case is listed for completeness, in terms of providing a catalogue of algorithms, and therefore a better tool set. It does not represent a significant area for research.

9.4 Summary of future work

This chapter has highlighted a number of areas for potential improvements for covering array generation. For practical use, the two areas of research that are of most interest are: how to handle varying numbers of values, and how to generate covering arrays deterministically for $d > 2$. In any practical situation, it is likely that the parameters will all have different numbers of values. Being able to select the level of interaction coverage is also important, as it gives better options for trade-offs of test coverage versus test budget.

Resolving the issues regarding the selection of block sizes would be the next most important area to investigate, as this could produce significant reductions in the number of configurations, but only in some cases. However, since a large test suite may be run for every single test configuration, even a small reduction in the number of configurations may have significant impact on the time and cost of system testing.

References

[Ad61] S. Addelman, O. Kempthorne, "Some main-effect plans and orthogonal arrays of strength two," *Annals of Mathematical Statistics*, Vol. 32, 1961, pp. 1167-1176.

[Bei90] B. Beizer, *Software Testing Techniques*, Second Edition, Van Nostrand Reinhold, New York NY USA, 1990.

[Ber] M. Berkelaar, "lp_solve," version 3.0. Linear programming solver developed at Eindhoven University, available at ftp://ftp.ics.ele.tue.nl/pub/lp_solve

[Bi00] R. Binder, *Testing of Object-Oriented Systems*, Addison-Wesley, Reading MA USA, 2000.

[Bor92] S.Yu. Boroday, I.S. Grunskii, "Recursive Generation of Locally Complete Tests," *Cybernetics and Systems Analysis*, Vol. 28, No. 4, July-August 1992, pp. 504-508.

[Bos38] R.C. Bose, "On the application of the properties of Galois fields to the construction of hyper Graeco-Latin squares," *Sankhya* Vol. 3, 1938, pp. 323-338.

[Bos52] R.C. Bose, K.A. Bush, "Orthogonal Arrays of Strength Two and Three," *Annals of Mathematical Statistics*, Vol. 23, 1952, pp. 508-524.

[Br92] R. Brownlie, J. Prowse, and M.S. Phadke, "Robust Testing of AT&T PMX/StarMail using OATS", *AT&T Technical Journal*, Vol. 71 No. 3, May/June 1992, pp. 41-47.

[Buh96] R.J.A. Buhr, R.S. Casselman, *Use Case Maps for Object-Oriented Systems*, Prentice-Hall, Upper Saddle River NJ USA, 1996.

[Buh98] R.J.A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, and S. Mankovski, "High Level, Multi-agent Prototypes from a Scenario-Path Notation: A Feature Interaction Example," *Proceedings of the Conference on Practical Application of Intelligent Agents and Multi-agent Technology (PAAM '98)*, London UK, March 1998. Available at http://www.sce.carleton.ca/ftp/pub/UseCaseMaps.

[Bur94] K. Burroughs, A. Jain, and R.L. Erickson, "Improved Quality of Protocol Testing Through Techniques of Experimental Design", *Proceedings of the IEEE International Conference on Communications (Supercomm/ICC '94)*, 1994, pp. 745-752.

[Bus52] K.A. Bush, "Orthogonal Arrays of Index Unity," *Annals of Mathematical Statistics*, Vol. 23, 1952, pp. 426-434.

[Cha99] M.A. Chateauneuf, C.J. Coulbourn, D.L. Kreher, "Covering arrays of strength three," *Designs, Codes, and Cryptography*, Vol. 16, 1999, pp. 235-242.

[Chv79] V. Chvatal, "A Greedy Heuristic for the Set-Covering Problem," *Mathematics of Operations Research*, Vol. 4, No. 3, August 1979, pp. 233-235.

[Coh94a] D.M. Cohen, S.R. Dalal, A. Kajla, and G.C. Patton, "The Automatic Efficient Test Generator (AETG) System," *Proceedings of the 5th International Symposium on Software Reliability Engineering (ISSRE '94)*, Monterey CA USA, 1994, pp. 303-309.

[Coh94b] D.M. Cohen, S.R. Dalal, A. Kajla, and G.C. Patton, "The AETG System for Feature and Protocol Testing", *Proceedings of the Third International Conference on Software Testing, Analysis, and Review*, Washington DC USA, June 1994.

[Coh96] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton, "The Combinatorial Approach to Automatic Test Generation," *IEEE Software*, vol. 13, no. 5, Sept. 1996, pp. 83-88.

[Coh97] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design," *IEEE Transactions on Software Engineering*, vol. 23, no. 7, July 1997, pp. 437-444.

[Col96] C.J. Colbourn, J. Dinitz, editors, *The CRC Handbook of Combinatorial Designs*, Chemical Rubber Company Press, 1996.

[Cr] P. Crescenzi, V. Kann, editors, "A compendium of NP-complete problems," available at http://www.f.kth.se/~viggo/problemlist/compendium.html.

[Din] J.H. Dinitz, "Handbook of Combinatorial Designs, New Results," On-line at http://www.emba.uvm.edu/~dinitz/newresults.html

[DS83] J.H. Dinitz and D. R. Stinson, "MOLS with holes," *Discrete Math.* Vol. 44 (1983), 145-154.

[Duh97] R.-C. Duh, M. Fürer, "Approximation of k-Set Cover by Semi-Local Optimization," in *Proceedings of the 29th Annual. ACM Symposium on Theory of Computing (STOC' 97)*, El Paso TX USA, 1997, pp. 256-264.

[Dun97] I.S. Dunietz, W.K. Ehrlich, B.D. Szablak, C.L. Mallows, A. Iannino, "Applying Design of Experiments to Software Testing," *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, Boston MA USA, 1997, pp. 205-215.

[Er99] H. Erdogmus, "Building a business case for COTS-centric development: an investment analysis perspective," in Summary of the First Workshop on Ensuring Successful COTS Development, Workshop at the 21st International Conference on Software Engineering (ICSE '99), Los Angeles CA USA, 1999. On line at http://wwwsel.iit.nrc.ca/projects/cots/icsewkshp/papers/erdogmus.pdf.

[Fe98] U. Feige, "A Threshold of ln n for Approximating Set Cover," *Journal of the ACM*, Vol. 45, No. 4, July 1998, pp. 634-652.

[Ga95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading MA USA, 1995.

[Gi76] W.J. Gilbert, *Modern Algebra with Applications*, Wiley Interscience, New York NY USA, 1976.

[Go95] D. M. Gordon, G. Kuperberg, O. Patashnik, "New Constructions for Covering Designs," *Journal of Combinatorial Designs*, Vol. 3, 1995, pp. 269-284.

[Hal86] M. Hall Jr., *Combinatorial Theory*, Wiley Interscience, New York, 1986.

[Har92] R.H. Hardin, N.J.A. Sloane, "Operating manual for Gosset: A General Purpose Program for Designing Experiments," Second Edition, Statistics Research Report No. 106, Bell Laboratories, Murray Hill NJ USA, October 1992.

[Har93] R.H. Hardin, N.J.A. Sloane, "A New Approach to the Construction of Optimal Designs," *Journal of Statistical Planning and Inference*, Vol. 37, 1993, pp. 339-369.

[He97] K. Heinrich, "Incomplete Transversal Designs," talk at the Workshop on Orthogonal Arrays and Transversal Designs, Kitchener-Waterloo ON, April 1997.

[Ho82] D.S. Hochbaum, "Approximation Algorithms for the Set Covering and Vertex Cover Problems," *SIAM Journal on Computing*, Vol. 11, No. 3, August 1982, pp. 555-556.

[ISO92] International Standards Organization (ISO), *OSI Conformance Testing Methodology and Framework — Part 3: The Tree and Tabular Combined Notation*, International Standard 9646-3, 1992.

[ITU00a] International Telecommunications Union (ITU-T), *Recommendation Z.120: Message Sequence Charts*, revised 2000.

[ITU00b] International Telecommunications Union (ITU-T), *Recommendation Z.100: Specification and Description Language*, revised 2000.

[Jo74] D.S. Johnson, "Approximation Algorithms for Combinatorial Problems," *Journal of Computer and System Sciences*, Vol. 9, 1974, pp. 256-278.

[Kar97a] M. Karpinski, "Polynomial Time Approximation Schemes for Some Dense Instances of NP-Hard Optimization Problems," in *Proceedings of the 1st Symposium on Randomization and Approximation Techniques in Computer Science*, Lecture Notes in Computer Science 1269, Springer-Verlag, 1997, pp. 1-14.

[Kar97b] M. Karpinski, A. Zelikovsky, "Approximating Dense Cases of Covering Problems," ECCC Technical Report TR97-004, 1997. Available at ftp://eccc.uni-trier.de/pub/eccc/reports/1997/TR97-004/index.html.

[Kat75] G.O.H. Katona, "Extremal problems for hypergraphs," in *Combinatorics*, edited by M. Hall Jr. and J.H. van Lint, Riedel Dordrecht, Holland, 1975, pp. 215-244.

[Kl73] D.J. Kleitman, J. Spencer, "Families of k-independent sets," *Discrete Mathematics*, Vol. 6, 1973, pp. 255-262.

[Le98] Y. Lei, K.C. Tai, "A test generation strategy for pairwise testing," in *Proceedings of the 3rd IEEE High-Assurance Systems Engineering Symposium*, 1998, pp. 254-261.

[LR79] J.K. Lenstra, A.H.G. Rinnooy Kan, "Complexity of packing, covering and partitioning problems," in: *Packing and covering in combinatorics*, Hrsg. Schrijver, A., Math. Centre Tracts 106, Amsterdam, 1979, pp. 275-291.

[Man85] R. Mandl, "Orthogonal Latin squares: An application of experiment design to compiler testing," *Communications of the ACM*, Vol. 28 No. 10, October 1985, pp. 1054-1058.

[Mar95] B. Marick, *The Craft of Software Testing: subsystem testing including object-based and object-oriented testing*, PTR Prentice-Hall, Englewood Cliffs NJ USA, 1995.

[Mc94] R. McDaniel, J.D. McGregor, "Testing the Polymorphic Interactions between Classes," Clemson University Department of Computer Science, Technical Report TR 94-103, Clemson SC USA, 1994.

[My79] G.J. Myers, *The Art of Software Testing*, John Wiley and Sons, New York NY USA, 1979.

[Ow] A. Owen, The "oa" orthogonal array generator program, Dept. of Statistics, Stanford University, Stanford CA USA. Available at http://fmad-www.larc.nasa.gov/mdob/users/jotto/Current_research/DOE/owen_prog_readme.html.

[Oz94] N. Ozmirzak, *A Knowledge-based environment for the Validation of Simulation Models*, Ph.D. Thesis, University of Ottawa, 1994.

[Pa91] C.H. Papadimitriou, "Optimization, Approximation, and Complexity Classes," *Journal of Computer and System Sciences*, Vol. 43, 1991, pp. 425-440.

[Pa98] C.H. Papadimitriou, K. Steiglitz, *Combinatorial Optimization: algorithms and complexity*, Dover, Mineola NY USA, 1998.

[Pr99] R.L. Probert, A.W. Williams, "Fast functional test generation using an SDL model," in *Proceedings of the 12th International Workshop on the Testing of Communicating Systems (IWTCS '99)*, Budapest Hungary, Sept. 1999, pp. 299-315.

[Pr01] R.L. Probert, H. Ural, A.W. Williams, "Rapid generation of functional tests using MSCs, SDL and TTCN," *Computer Communications*, Vol. 24, No. 3-4, February 15, 2001, pp. 374-393.

[Ra47] C.R. Rao, "Factorial Experiments Derivable from Combinatorial Arrangements of Arrays," *Journal of the Royal Statistical Society*, Vol. 9, No. 1, 1947, pp. 128-139.

[Ro96] P.J. Ross, *Taguchi Techniques for Quality Engineering*, Second edition, McGraw-Hill, New York NY USA, 1996.

[Ry00] J. Ryu, M. Kim, S. Kang and S. Seol, "Interoperability Test Suite Generation for the TCP Data Part Using Experimental Design Techniques," in *Proceedings of the 13th International Conference on the Testing of Communicating Systems (Testcom 2000)*, Ottawa ON, August 2000, pp. 127-142.

[Sa00] K. Saleh, H. Ural, A. Williams, "Test generation based on control and data dependencies within system specifications in SDL", *Computer Communications*, Vol. 23, March 15, 2000, pp. 609-627.

[Se94] B. Selic, G. Gullekson, P.T. Ward, *Real-time Object-Oriented Modeling*, John Wiley & Sons, New York NY USA, 1994.

[Sh94] S.B. Sherwood, "Effective Testing of Factor Combinations," *Proceedings of the Third International Conference on Testing Software*, Washington DC USA, June 1994.

[Sl93] N.J.A. Sloane, "Covering Arrays and Intersecting Codes," *Journal of Combinatorial Designs*, Vol. 1, 1993, pp. 51-63.

[Sl97] N.J.A. Sloane, "Linear Programming Bounds for Orthogonal Arrays," talk at the Workshop on Orthogonal Arrays and Transversal Designs, Kitchener-Waterloo ON, April 1997.

[Sr95] A. Srinivasan, "Improved Approximation of Packing and Covering Problems," in *Proceedings of the 27th Annual. ACM. Symposium on Theory of Computing (STOC' 95)*, Las Vegas NV USA, 1995, pp. 268-276.

[St98a] B. Stevens, *Transversal Covers and Packings*, Ph.D. thesis, University of Toronto, 1998.

[St98b] B. Stevens, E. Mendelsohn, "Efficient Software Testing Protocols," *Proceedings of the 8th IBM Centre for Advanced Studies Conference (CASCON '98)*, Toronto ON, 1998, pp. 279-293.

[St98c] B. Stevens, L. Moura, E. Mendelsohn, "Lower Bounds for Transversal Covers," *Designs, Codes, and Cryptography*, Vol. 15, 1998, pp. 279-299.

[St00] B. Stevens, in seminar given at the University of Ottawa, December 2000.

[Su] Sun Microsystems Inc., Java 2 Platform, Standard Edition, Version 1.3 API Specification. Sun Microsystems, 2000. Available on-line at http://java.sun.com/j2se/1.3/docs/api

[Sy] Symantec Corp., Symantec C++ Development Environment for the Macintosh, version 7.0, Symantec Corp., Cupertino CA USA.

[Ta87] G. Taguchi, *System of Experimental Design, Vol. 1 and 2*, UNIPUB/Kraus International Publications, 1987.

[Te] Telelogic Tau tool set, version 3.3, Telelogic AB, Malmö Sweden.

[To] R. Tobias, "SAS® Macros for Orthogonal Array Constructions, " SAS Institute Inc., Cary NC USA, available at http://lib.stat.cmu.edu/designs/oa.SAS

[Ur93] H. Ural, and A.W. Williams, "Test generation by exposing control and data dependencies within system specifications in SDL", *Proceedings of the 6th International Conference on Formal Description Techniques (FORTE '93)*, Cambridge MA USA, Oct. 1993, pp. 339-354.

[Wh96] L.J. White, "Regression Testing of GUI Event Interactions", *Proceedings of the International Conference on Software Maintenance*, Monterey CA, Nov. 1996, pp 350-358.

[Wi96] A.W. Williams, R.L. Probert, "A Practical Strategy for Testing pair-wise Coverage of Network Interfaces," *Proceedings of the 7th International Conference on Software Reliability Engineering (ISSRE '96)*, White Plains NY USA, October 1996, pp. 246-254.

[Wi00] A.W. Williams, "Determination of Test Configurations for Pair-Wise Interaction Coverage," in *Proceedings of the 13th International Conference on the Testing of Communicating Systems (Testcom 2000)*, Ottawa ON, August 2000, pp. 59-74.

[Wi01] A.W. Williams, R.L. Probert, "A Measure for Component Interaction Test Coverage," to appear in *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2001)*, Beirut Lebanon, June 2001.