4

DETERMINATION OF TEST CONFIGURATIONS FOR PAIR-WISE INTERACTION COVERAGE

Alan W. Williams[§]
School of Information Technology
University of Ottawa
Ottawa ON K1N 6N5, Canada
awilliam@site.uottawa.ca

Abstract Systems constructed from components, including distributed systems, consist of a number of elements that interact with each other. As the number of network elements or interchangeable components for each network element increases, the trade off that the system tester faces is the *thoroughness of test configuration coverage*, versus availability of limited resources (time and budget). An approach to resolving this trade off is to determine a minimal set of test configurations that test each pair-wise combination of components. This goal gives a well-defined, cost-effective level of test coverage, with a reduced number of system configurations. To select such a set of test configurations, we show how to apply the method of covering arrays, and improve on previous results.

Keywords: System testing, Test coverage, Interactions, Component-based testing

1. INTRODUCTION

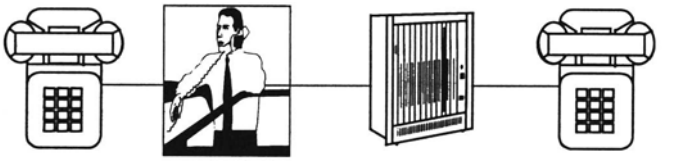
A common source of system faults is unexpected interactions between system components. The risk is magnified when there are a number of interchangeable components for each element in a system. A manufacturer of these system components would want to test as many of the potential system

[§] The author gratefully acknowledges the support of: the IBM Centre for Advanced Studies, Nortel Networks Ltd., Mitel Corporation, ObjecTime Ltd., CITO, NSERC, and the University of Ottawa.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35516-0_20

H. Ural et al. (eds.), *Testing of Communicating Systems*

configurations as possible, to reduce the risk of interaction problems. However, the number of potential system configurations grows exponentially. The scenario in Figure 1 has four parameters, each with three values. There are $3^4 = 81$ possible configurations.



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

Figure 1. A System Test Scenario

A system tester faces the constraints of time and money, and all possible configurations cannot be tested within any reasonable allotment. For each system test configuration, a suite of system tests must be executed. Changing between configurations normally requires additional effort. Therefore, reducing the number of system test configurations will produce significant savings. How, then, can the risk of interaction faults be managed in a realistic test plan?

One approach is to at least test for all two-way interactions among various system components. This leads to a reduced set of test configurations. The assumption is that the risk of an interaction among three or more components is balanced against the ability to complete system testing within a reasonable budget. This paper investigates this approach, using covering arrays to determine the set of test configurations that cover all two-way interactions.

This paper starts with an overview of design of statistical experiments, and then shows how to apply these concepts to the testing of software. Orthogonal arrays and covering arrays are defined. An algorithm to generate orthogonal arrays is presented, and these are used to derive building blocks for constructing larger covering arrays. An algorithm, improving on the results from [14], for larger covering arrays is presented, first using an example, and then in general. Algorithm results are provided with respect to the number of configurations produced. The paper concludes with a summary, and opportunities for further work.

2. APPLYING STATISTICAL EXPERIMENTAL DESIGN TO SOFTWARE TESTING

There is a great deal of work in the literature of statistics, devoted to the subject of experimental design. Except for a few instances ([2], [3], [4], [5], [6] [9], [10]), the results have not generally been applied to the testing of software. Areas where these techniques are often used to detect interactions are in chemical, biological, and medical research. They are increasingly being advocated for use in engineering quality control.

We can make excellent use of these techniques when testing software. The software testing "experiment" that is proposed is to determine if all of the software components are truly free of unwanted interactions. A successful run of the experiments would confirm this independence. A single experiment would consist of running the entire set of test cases for an application, over a specific test configuration. (The prior existence of a suitable "complete" test suite is assumed.)

The application of this method 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.

There are two considerations that 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 components. 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 range, as would be the case with traditional experimental design.

Second, a statistical analysis of real-valued results is not required. The expected test results are determined directly (e.g. from specifications or customer requirements.) After running each test case, the result is a test verdict: one of "pass," "fail," or perhaps "inconclusive." One experiment corresponds to running the entire test suite, 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, running the test suite 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 analyse a set of real-valued results, with confidence intervals and so on.

Therefore, the requirement for balance can be removed. If every possible pair-wise combination of values is covered, any interactions between those

values can be detected. Removing the need for balance provides for a significant reduction in test configurations required.

One possible goal is to determine 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 [4] show that covering two-way interactions results in excellent code coverage. When required, experiments can be designed to find interactions among three or more specific parameter values, at the expense of an increased number of configurations. This paper will focus on pair-wise parameter value coverage.

3. ORTHOGONAL AND COVERING ARRAYS

In this section, we first set up the test configuration problem. Next, orthogonal arrays and covering arrays are defined. An algorithm is provided for generating orthogonal arrays. Based on these arrays, we introduce "basic" and "reduced" arrays, which can be used as building blocks to build larger covering arrays.

3.1 The Configuration Problem

Suppose there are k independent system parameters. The property of independence means that the selection of a particular value for one parameter does not effect the selection of any other parameter values. (See [14] for how to handle dependencies among parameters.) Each parameter has a set of discrete values that the parameter can take. Suppose parameter i has n_i values, which are enumerated as $1 \ldots n_i$. Without loss of generality, assume that the parameters are ordered so that $n_1 \geq n_2 \geq \ldots \geq n_k$.

A test configuration consists of a selection of values for each parameter. Therefore, it consists of a k-tuple of parameter values. The goal is to minimise the number of k-tuples that achieve a test coverage criterion.

We also introduce a value $n = f(n_1,\ldots,n_k)$ that is a scalar representation for the "number of values" in various array constructions. Aside from conversion to a scalar value, the reason for introducing a separate variable is that some of the constructions we shall use have additional constraints on n. These constraints can be incorporated into the function f.

3.2 Definitions

A standard construction used for the design of statistical experiments is known as an orthogonal array.

Let $O(c,k,n,t)$ denote an orthogonal array, where:

- c is the number of rows in the array. The k-tuple forming each row is a single test configuration, and thus c is 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 $1,\ldots,n$, where $n = f(n_1,\ldots,n_k)$. Typically, this means that each parameter would have (up to) n values.
- t is the strength of the array (see below).

An orthogonal array has *strength* t if in any $c \times t$ sub-matrix (that is, select any t columns), each of the n^t possible t-tuples (rows) appears the same number of times (>0). In other words, all t-way combinations of parameter values occur the same number of times. It is this balance property that defines the orthogonal array. This property allows the effect of a single parameter to be detected, as well as interactions among (up to) t parameters.

The definition of covering array (see [12] or [13]), is identical to the definition of orthogonal array, except that the strength of the array is redefined as follows:

A *covering array*, denoted $C(c,k,n,t)$, has *strength* t if in any $c \times t$ sub-matrix (that is, selecting t columns), each of the n^t possible t-tuples (rows) appear at least once. In other words, all t-way combinations of parameter values occur at least once. This definition of strength captures the desired t-way interaction property.

This compares with an orthogonal array, where each t-interaction elements appears at least once, and also the same number of times. An orthogonal array satisfies the requirements of a covering array, but a covering array may not be orthogonal. Table 1 shows examples of an orthogonal array (left) and a covering array (right). In the orthogonal array, you can select any two columns, and each ordered pair (1,1), (1,2), (2,1), and (2,2) appears exactly once. In the covering array, each of those ordered pairs will occur at least once, but some may be repeated

Table 1. Examples of an orthogonal array (left) and a covering array (right)

1	1	1
1	2	2
2	1	2
2	2	1

1	1	1	1
2	2	2	1
2	2	1	2
2	1	2	2
1	2	2	2

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. The ability to analyse real-value results is also lost, but this is not required when test verdicts are derived independently. Therefore, the covering array preserves all of the properties needed for software testing, and we can reduce the number of configurations required, as compared with an orthogonal array.

For the rest of the paper, unless explicitly specified, a default value of $t = 2$ is assumed, and the orthogonal and covering arrays will be denoted as $O(c,k,n)$, and $C(c,k,n)$, respectively. This corresponds to the testing goal of pair-wise coverage of parameter values.

3.3 Orthogonal Array Construction

Orthogonal arrays do not necessarily exist for arbitrary values of c, k, and n (see [11] for restrictions). We will use a construction for arrays of strength 2 adapted from Bose [1] for $O(n^2,n+1,n)$. In this construction, n is required to be a prime power. Therefore, we shall define function $n = f(n_1,\ldots,n_k)$ such that $n \geq n_1$ and $n = p^m$ for some prime number p and integer $m \geq 1$. The value of n is the largest number of parameter values, increased to the next highest prime power. In this particular case, the value of f is only dependent on n_1, and not on any of the other parameters.

Let $GF(n) = \{x_0,x_1,\ldots,x_{n-1}\}$ be a Galois (finite) field of order n where $x_0 = 0$ and $x_1 = 1$. If the value of n is a prime number, then the integers modulo n can be used. For values where n is a prime power, see, for example, [7] for an algorithm to generate a Galois field.

To find the element of the array $O_{ij}(n^2,n+1,n)$ for $i = 1,\ldots,n^2$, $j = 1,\ldots,n+1$, calculate $u = \lfloor (i-1)/n \rfloor + 1$ and $w = ((i-1) \bmod n) + 1$. Then, $O_{i1}(n^2,n+1,n) = u$, $O_{i2}(n^2,n+1,n) = w$. When $j > 2$, determine q such that, using the Galois field arithmetic, $x_q = x_{j-2} \cdot x_u + x_w$, and $O_{ij}(n^2,n+1,n) = q$.

This algorithm produces orthogonal arrays that have several useful properties, which will be taken advantage of in subsequent constructions:

- The first row is all ones.
- The first n rows have entries equal to the row index, excepting the first column.
- The first column consists of n ones, n twos, etc.
- Each column of $O(n^2,n+1,n)$, except the first, is comprised of sets of permutations of $(1,\ldots,n)$.

In Figure 1 (in section 1), a scenario was illustrated where there are four system parameters, each with three possible values. There are 81 possible

configurations, but pair-wise coverage can be achieved with only 9 configurations.

3.4 Building Blocks for Covering Arrays

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.

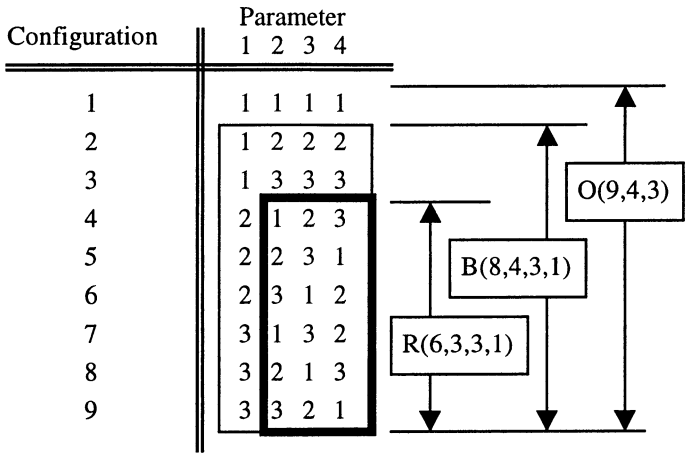


Figure 2. Orthogonal (O), Basic (B), and Reduced (R) arrays

Define a "basic" array $B(n^2-1,n+1,n,d) = O(n^2,n+1,n)$ with the first row removed, and each column duplicated d times consecutively. The array $B(n^2-1,n+1,n,d)$ has $n^2 - 1$ rows and $(n + 1) \times d$ columns.

Define a "reduced" array $R(n^2-n,n,n,d) = O(n^2,n+1,n)$ with the first n rows removed, the first column removed, and each (remaining) column duplicated d times consecutively. $R(n^2-n,n,n,d)$ has $n^2 - n$ rows and $n \times d$ columns.

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

The general strategy for using the basic and reduced arrays is that the reduced array has fewer rows, so it contributes fewer test configurations. However, with fewer columns, the reduced array has less parameter capacity. The basic array will be used only when the reduced array cannot hold enough parameters.

We now introduce some building blocks that are not based on an orthogonal array. However, they are useful for constructing larger covering arrays, as we shall see. Let $I(c,d)$ denote an array that contains all ones, and is c rows by d columns. Let $N(n^2-n,n,d)$ denote an array that contains a $n \times d$

block of twos, vertically concatenated with a $n \times d$ block of threes, and so on up to n. $N(c,n,d)$ has c rows and $n \times d$ columns. These constructions are used to add parameter capacity when possible.

For illustrative purposes, the notation in Table 2 will be used to show how building blocks are combined to form larger arrays

Table 2. Notation for large array construction

O(9,4,3)	O(9,4,3)	O(9,4,3)	I(9,1)
R(6,3,3,4)			N(6,3,1)

Table 2 indicates that three copies of $O(9,4,3)$, and then I(9,1) are to be concatenated horizontally. Below the three copies of $O(9,4,3)$, the array $R(6,3,3,4)$ is to be concatenated vertically, and below the array I(9,1), the array N(6,3,1) is to be concatenated. The resulting large array has 13 columns and 15 rows.

4. CONSTRUCTING LARGER COVERING ARRAYS

In this section, we show how to construct covering arrays that are not subject to the restrictions on the number of parameters that apply to orthogonal arrays.

The orthogonal array $O(n^2,n+1,n)$ is constrained to (a maximum limit of) $k = n + 1$ parameters [11]. 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 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.

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 $O(9,4,3)$ (see Table 2). 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.

Consider the first column of the first copy of $O(9,4,3)$. Because of the orthogonal array construction, pair-wise coverage is achieved with the remaining three columns in that first copy of $O(9,4,3)$. Each of the additional copies of $O(9,4,3)$ has duplicates of those same three other columns. The

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

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. The columns of R(6,3,3,1) can be inserted under each of the first columns of the O arrays.

It turns out that this process can be repeated for the each column in the first repetition of $O(9,4,3)$. The effect is to duplicate the columns of $R(6,3,3,1)$, effectively forming $R(6,3,3,4)$.

Since we have expanded the array in the vertical direction, there is now capacity for a thirteenth parameter. We can add a column of nine 1's (that is, $I(9,1)$) to the right of the O arrays, and the values 2 to n (repeated n times each vertically) next to the R array (that is, $N(6,3,1)$).

Why does this work? We are making use of the properties of the Bose orthogonal array construction. In the R array, we have $n - 1$ permutations of $1,\ldots,n$ in each column. Therefore, we can construct a column of n 2's, n 3's, and so on up to n. This covers all combinations in the other columns except for the $(1,x)$ combinations. We can then fill the part of the column next to the O arrays with 1's. Except for the first column of the O arrays (and their identical counterparts), the O array columns are permutations of $1,\ldots,n$. The first columns are structured as n 1's followed by n 2's and so on up to n's. Filling the column with 1's will definitely achieve pair-wise coverage.

In fact, we need to fill the column with n 1's at the start, and then in every nth position after that. The remainder of the positions could be left as "don't care" values, where the value of the parameter does not affect the final coverage. It could be set to any legal value to construct a test configuration.

Because this step results in "don't care" values and an unbalanced use of values (the "1" in the example), the tester can judiciously choose the correspondence between the enumerated values $1,2,\ldots,n$ 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 "1" in the extra column, and fill in the "don't care" values with ones 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 1's, to form a column of n^2 1's.

Pair-wise coverage of all parameters has now been achieved. A notation for this *two-stage* construction of C(15,13,3) is shown in Table 3.

Table 3. Construction of C(15,13,3)

Stage 1	O(9,4,3)	O(9,4,3)	O(9,4,3)	I(9,1)
Stage 2	R(6,3,3,4)			N(6,3,1)

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 $O(n^2,n+1,n)$." 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 $O(n^2,n+1,n)$ 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.

Using the reduced array instead of the basic array means that we cannot add an additional column. The basic array does not have the property that its columns are consecutive permutations of $1,\ldots,n$.

For example, if there are sixteen parameters, each with three values, we can construct $C(17,16,3)$ as shown in Table 4.

Table 4. $C(17,16,3)$ in shorthand notation

$O(9,4,3)$	$O(9,4,3)$	$O(9,4,3)$	$O(9,4,3)$
$B(8,4,3,4)$			

If there are more than $(n + 1)^2$ parameters, the entire process can be repeated recursively, both horizontally and vertically.

The extra column presents some additional difficulties for additional stages. However, we can replicate the extra columns as we did with the rest of the array, and then cover the missing combinations. It turns out that we can just use the same method of covering the missing combinations of identical columns: add a reduced array underneath the duplicated columns. We can then add a final extra column, again because of the increased vertical dimension. An example of a three-stage construction is shown Table 5.

Table 5. A three-stage construction of covering array $C(21,40,3)$

O	O	O	O	O	O	O	O	O	I(9,3)	I(15,1)
R(6,3,3,4)			R(6,3,3,4)			R(6,3,3,4)			N(6,3,3)	
R(6,3,3,12)									R(6,3,3,1)	N(6,3,1)

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

4.1 Algorithms for Covering Array Construction

To construct a covering array of strength 2, we are given k, the number of parameters, and that parameter i has n_i values, which are enumerated as $1,\ldots,n_i$. Let n be the least integer where $n \geq n_1$ and $n = p^m$ for some prime number p and integer $m \geq 1$. The number of stages required is $s = \lceil \log_{n+1}(k) \rceil$. If $s = 1$, then a subset of k columns of $O(n^2,n+1,n)$ can be used as $C(c,k,n)$, and it is not necessary to proceed further.

4.1.1 Determining the building blocks to use

Let g_r represent the number of columns to be used in the arrays at stage r. Specifically, $g_r = n + 1$ indicates that at stage r, copies of the basic array are used. If $g_r = n$, then copies of the reduced array are used. The initial number of columns for arrays in each stage is $g_1 = n + 1$ (representing the use of the orthogonal array $O(n^2,n+1,n)$ in the first stage), and $g_r = n$ for $r = 2,\ldots,s$. We are hoping to use reduced arrays for all stages after the first one.

Algorithm 1. Determine parameter capacity

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

Parameter capacity $h \leftarrow n + 1$
For $r = 2,\ldots,s$:
    $h \leftarrow h \times g_r$
    If $g_r = n$ then $h \leftarrow h + 1$ (for extra column)

Algorithm 1 is of order $\log(k)$, since the for loop will execute up to s-1 times.

Algorithm 2. Determine usage of basic versus reduced arrays

$g_1 \leftarrow n + 1$, and $g_r \leftarrow n$ for $r = 2,\ldots,s$
$r \leftarrow 2$.
Determine parameter capacity h.
While $h < k$:
    $g_r \leftarrow n + 1$; increment r
    Redetermine h

Algorithm 2 is of order $\log(k)$, since the for and while loops will execute up to s-1 times. After using Algorithm 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.

Algorithm 3. Determine column duplication values

The next calculation is the values to use for d in the basic or reduced arrays at each stage. Let d_r represent the column duplication value at stage r. Then, $d_1 = 1$, and $d_r = d_{r-1} \times g_{r-1}$ for $r = 2,\ldots,s$. Algorithm 3 is of order $\log(k)$, since there are s values of the d_r.

Algorithm 4. Values to set up "extra" columns

For each stage that uses reduced arrays, we need to determine the parameters for the extra columns. Let e_r be the number of extra columns in stage r. Thus, at stage r, the arrays $I(c,e_r)$ and $N(n^2-n,n,e_r)$ are used, where c is the current number of rows in the covering array. Then, $e_r = 0$ if $g_r = n + 1$, or $e_r = (d_s\, g_s) / (d_r\, g_r)$ if $g_r = n$

Then, 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. Let λ_{rq} be the duplication factor for the reduced array added at stage q, for the extra columns added at stage r. For the columns added at any particular stage, the duplication factor starts at 1 in the next stage, and is multiplied by n for subsequent stages. $\lambda_{rq} = 1$ if $q = r + 1$ and $r + 1 \le s$, and $\lambda_{rq} = \lambda_{r,q-1} \times n$ for $q = r + 2,\ldots,s$

Calculation of the values for e_r is of order $\log(k)$, since there are s values of the e_r. Calculation of the λ_{rq} is of order $\log^2(k)$, since there are (up to) s values of r and q.

4.1.2 Building the covering array

The results of the preceding section give us the structure of the building blocks that are needed to construct $C(c,k,n,2)$. This section describes how to assemble the building blocks.

It is assumed that if an array has a duplication factor of 0, it is a null array, and concatenating a null array results in the original array.

Algorithm 5. Construct $C(c,k,n,2)$

The following algorithm builds $C(c,k,n,2)$:

Start with C as $O(n^2,n+1,n)$ repeated $(d_s\, g_s) / (n + 1)$ times horizontally
For $r = 2,\ldots,s$:
    Add to the right of C, $I(c,e_r)$; c is current number of rows in C.
    Let A be $R(n^2-n,n,n,d)$ if $g_r = n$, or $B(n^2-1,n+1,n,d)$ if $g_r = n + 1$,
    Construct array S as A repeated $(d_s\, g_s) / (d_r\, g_r)$ times horizontally
    For $q = 2\ldots r$:
        Add to the right of S $e_r / (n\lambda_{rq})$ copies of $R(n^2-n,n,n,\lambda_{rq})$

Horizontally concatenate array $N(n^2-n,n,e_r)$ to the right of S
Vertically concatenate array S below C

Overall, algorithm 5 is of order $k\log^2(k)$. The nested r and q loops are executed on the order of $\log^2(k)$ times. The horizontal concatenations are repeated order k times as the largest number of repetitions is k/n times.

Because the algorithms are executed in sequence, the overall complexity is on the order of $n^2 + k\log^2(k)$. Construction of the O, R, and B arrays is of order n^2. This is done prior to the execution of algorithm 5, so the complexities are added.

5. ALGORITHM RESULTS

5.1 Results for the Example

For the example in Figure 1 (in section 1), Table 6 shows the test configurations determined from C(9,4,3) that provide pair-wise parameter value coverage for this situation:

Table 6. Test configurations for the scenario in Figure 1

#	Caller	Type	Market	Called
1:	Regular	Local	Canada	Regular
2:	Regular	Long distance	US	Cell phone
3:	Regular	Toll free	Mexico	Pager
4:	Cell phone	Local	US	Pager
5:	Cell phone	Long distance	Mexico	Regular
6:	Cell phone	Toll free	Canada	Cell phone
7:	Coin phone	Local	Mexico	Cell phone
8:	Coin phone	Long distance	Canada	Page
9:	Coin phone	Toll free	US	Regular

5.2 Number of Configurations Generated

With regard to the number of test configurations for k parameters, with values $n_1 \ldots n_k$, the best case lower bound on the number of configurations produced is $\lceil \log_{n+1}(k) \rceil (n^2 - n) + n$ where n is the next integer $\geq n_1$ that is a prime power. The worst case upper bound is $\lceil \log_{n+1}(k) \rceil (n^2 - 1) + 1$. In general, the number of configurations is proportional to the square of the number of values, and the logarithm of the number of parameters.

Figure 3 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.

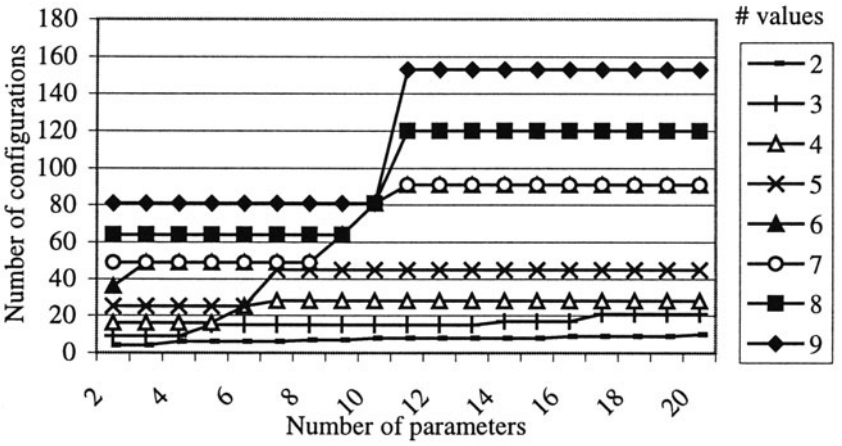


Figure 3. Results from the algorithm

5.3 Comparison with the IPO Algorithm

The algorithms presented here have been implemented in a tool (versions exist in C++ and Java) called "TConfig". For comparison, the TConfig tool was run using the situations described in the paper by Lei and Tai [8] on their in-parameter-order (IPO) strategy. This is a heuristic strategy that starts with two parameters, and configurations for each combination of the values for those two parameters. The strategy is then to grow the array horizontally by adding a single parameter at a time, and then growing the array vertically by adding test configurations as needed. Table 7 shows the run times and the number of configurations generated by each method. To ensure a fair comparison, the IPO strategy was implemented using the same compiler and class library, and the algorithms were run on the same machine. In fact, the IPO results presented here are slightly better than those reported in [8].

The complexity of the IPO algorithm is $n^5 \times k^3$ [8]. The algorithm presented here has complexity of $n^2 + k\log^2(k)$, which is a considerable improvement.

In all cases, because the algorithm presented here is deterministic, it runs much faster than the IPO method. For situations where the number of parameters are the same for each value, TConfig almost always generated fewer configurations to achieve pair-wise parameter value coverage; the exception is case 6. For situations where the numbers of values for each parameter differ, TConfig generated slightly more configurations.

Overall, the TConfig tool clearly outperforms the IPO method for execution time, and produced fewer test configurations in 13 of the 16 cases.

Table 7. Comparison with the IPO method

Case #	System	IPO		TConfig	
		# config	Time(s)	# config	Time (s)
1	4 parameters, 3 values	10	0.06	9	0.05
2	13 parameters, 3 values	20	0.44	15	0.05
3	61 parms: 15 × 4 values, 17 × 3 values, 29 × 2 vals.	34	2.75	40	0.05
4	75 parms: 1 × 4 values, 39 × 3 values, 35 × 2 values	27	3.90	30	0.05
5	100 parameters, 2 values	15	3.52	14	0.05
6	20 parameters, 10 values	219	47.46	231	0.44
7	10 parameters, 4 values	31	0.11	28	0.05
8	20 parameters, 4 values	34	0.99	28	0.05
9	30 parameters, 4 values	41	2.42	40	0.05
10	40 parameters, 4 values	42	5.00	40	0.05
11	50 parameters, 4 values	47	9.51	40	0.05
12	60 parameters, 4 values	47	15.98	40	0.05
13	70 parameters, 4 values	49	25.43	40	0.05
14	80 parameters, 4 values	49	38.50	40	0.05
15	90 parameters, 4 values	52	53.72	43	0.06
16	100 parameters, 4 values	52	75.02	43	0.11

6. CONCLUSIONS AND FURTHER WORK

This paper has shown that methods from statistical experimental design can be applied to the problem of determining a set of system test configurations that meet a defined goal for coverage of potential 2-way interactions. We presented improvements in algorithms for generating covering arrays. These covering arrays are constructed to ensure that pair-wise coverage of software components is achieved.

Further work needs to be done to address the situation where there are varying numbers of values for each parameter. A potential approach is to combine this approach with a heuristic such as the IPO method to deal with the differing number of values for each parameter.

7. ACKNOWLEDGMENTS

I would like to thank Dr. Robert Probert, and everyone in the Telecommunications Software Engineering Research Group at the University of Ottawa for their support and feedback on this work.

REFERENCES

[1] R.C. Bose. On the application of the properties of Galois fields to the construction of hyper Graeco-Latin squares. Sankhya 3:323-338 (1938).

[2] R. Brownlie, J. Prowse, and M.S. Phadke. Robust Testing of AT&T PMX/StarMail using OATS. AT&T Technical Journal, 71(3): 41-47 (May/June 1992).

[3] K. Burroughs, A. Jain, and R.L. Erickson. Improved Quality of Protocol Testing Through Techniques of Experimental Design. In Proceedings of Supercomm/ICC '94, 1994, pp. 745-752 1994.

[4] 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, 23(7): 437-444, (July 1997).

[5] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The Combinatorial Approach to Automatic Test Generation. IEEE Software, 13(5): 83-88, (September 1996).

[6] I.S. Dunietz, W.K. Ehrlich, B.D. Szablak, C.L. Mallows, A. Iannino. Applying Design of Experiments to Software Testing. In Proceedings of ICSE '97, pages 205-215, Boston MA USA, (1997).

[7] W.J. Gilbert. Modern Algebra with Applications. Wiley Interscience, New York NY USA, 1976.

[8] Y. Lei, K.C. Tai. A test generation strategy for pairwise testing. In Proc. of the 3rd IEEE High-Assurance Systems Engineering Symposium, pages 254-161, Nov. 1998.

[9] R. Mandl. Orthogonal Latin squares: An application of experiment design to compiler testing. Communications of the ACM, 28(10): 1054-1058 (October 1985).

[10] W.B. Perkinson. A Methodology for Designing and Executing ISDN Feature Tests Using Automated Test Systems. In Proceedings of IEEE GLOBECOMM '92, 1992.

[11] C.R. Rao. Factorial Experiments Derivable from Combinatorial Arrangements of Arrays. Journal of the Royal Statistical Society, 9(1): 128-139 (1947).

[12] B. Stevens, Transversal Covers and Packings, Ph.D. thesis, University of Toronto, 1998.

[13] B. Stevens, E. Mendelsohn. Efficient Software Testing Protocols. In Proceedings of CASCON '98, pages 279-293, Toronto ON Canada, December 1998.

[14] A.W. Williams, R.L. Probert, A Practical Strategy for Testing pair-wise Coverage of Network Interfaces. In Proc. of the 7th International Conference on Software Reliability *Engineering (ISSRE '96),* pages 246-254, White Plains NY USA, October 1996.