Applying Design of Experiments to Software Testing

Experience Report

I. S. Dunietz

W. K. Ehrlich

B. D. Szablak

AT&T Laboratories

Room 2C-265

700 Mountain Avenue

Murray Hill, NJ 07974 USA

+1 908 582 3851
clm@research.att.com

C. L. Mallows

AT&T NCS OTC

480 Red Hill Road

Middletown, NJ 07748 USA

A. Iannino

Pipeline Associates

9 Entin Road

Parsippany, NJ 07054 USA

+1 201 428 1700
ai@powerpage.com

ABSTRACT

Recently, a class of experimental designs has been devised that guarantee input domain coverage up to all combinations of \( k \) test factors taken \( t \) at a time. With such designs, all pairwise combinations (or triplets or quadruplets, etc.) are selected at least once. To evaluate their applicability to software testing, we analyzed the extent to which software coverage (i.e., code execution) achieved by these designs for \( t = 1,\ldots,k \) is representative of that achieved by exhaustively testing all factor combinations. The block coverage obtained for \( r \leq 2 \) was comparable with that achieved by exhaustively testing all factor combinations but higher-order values of \( t \) were required for path coverage. Implications of these results for software testing are discussed.

Keywords

Design of Experiments (DOE), partitioning, software testing

INTRODUCTION

For a Software Unit under Test (SUT), the set of all possible inputs (the input domain) is typically too large to test exhaustively. For example, consider a screen that contains 20 fields that accept character strings of different lengths (see Figure 1). If we assume that a user can enter one of 95 ASCII possibilities for each character within a field, and that there is a total of 151 characters that can be entered across all 20 screen fields, then there are \( 95^{151} \approx 4.3 \times 10^{398} \) different inputs or test cases corresponding to the different ways a user can populate the screen. Clearly, exhaustive input testing is not feasible and hence, a software tester needs to efficiently generate an effective set of test cases as a means of verifying the correct operation of the software.

Figure 1. TICKET OPEN/UPDATE screen with 20 fields of different field lengths.

One approach to test case generation is through the Design of Experiments (DOE) and statistical sampling. Here, the objective is to systematically "cover" the input domain of the SUT as efficiently as possible. Software engineers intimately familiar with the functionality of the SUT conceptualize a set of factors or parameters that are relevant to the SUT processing and that can be varied during testing. Each parameter or factor takes on some number of distinct settings, where a setting represents a possible range of values that can be assumed to be homogeneous. The product of these factors' settings defines a partition of the SUT's input domains. Inputs are selected for testing by sampling some number of inputs from each factor combination, or, alternatively, from each cell of the partition.
More formally, let \( i = 1, 2, ..., k \) denote factors relevant to the SUT processing, where each factor, \( i \), takes on \( q_i \) distinct settings. The product of these factors’ settings,

\[
Q = \prod_{i=1}^{k} q_i
\]

defines a partition of the SUT’s input domain. A total of \( n \) inputs is allocated for testing, with \( n_j \) inputs selected randomly from each cell of the partition, \( \sum_{j=1}^{Q} n_j = n \).

Typically, a constant sample allocation scheme is used in which the same number of inputs (usually equal to 1) is sampled from each cell of the partition, \( n_j = 1, j = 1, 2, ..., Q \).

In summary, a full factorial design that selects test cases (or inputs) from each cell of the partition serves as a tool to explore the input domain systematically. Note that this approach is related to “partition testing” \([14, 15]\) in which the objective is to divide or partition the SUT’s input domain into subsets or subdomains such that the partition is revealing of software faults.

**APPLICABILITY OF INCOMPLETE DESIGNS TO PARTITION TESTING**

When there are too many factors or factor levels, a full factorial design may lead to so many partition cells that the number of inputs (test cases) to be selected is practically infeasible. For example, for our 20-field screen, if we consider data entry within each field to represent a test factor, and if we assume 3 settings per test factor, Null Entry, Valid Entry, and Invalid Entry, then there are \( 3^{20} = 3,486,784,401 \) factor combinations. Assuming 1 input per factor combination, then there are approximately \( 3.5 \times 10^9 \) inputs to test. Although this number of inputs achieved through exhaustive partition testing represents a tremendous improvement over the number of inputs required by exhaustive input testing (\( 3.5 \times 10^9 \) vs. \( \sim 4.3 \times 10^{296} \)), complete coverage of all possible combinations of test factor settings is not feasible. Consequently, the test problem is to choose a subset of \( n \) cells, \( n \ll Q \), that “cover” the input space as effectively as possible.

**t-Factor Covering Designs**

Recently, a class of “incomplete” experimental designs that guarantee factor coverage up to all combinations of \( k \) test factors taken \( t \) at a time (i.e., \( t \)-factor covering designs \([2, 3, 4]\)) has been advanced as an efficient approach to test design. These designs can be described as follows.

Let \( (n, Q) \) denote a matrix of dimensions \( n \times k \) representing a design with \( i = 1, 2, ..., k \) factors and \( n \) runs. Each factor takes on \( q_i \) distinct levels so that \( Q = \prod_{i=1}^{k} q_i \). Note that \( n \leq Q \) represents the number of unique factor combinations (i.e., cells of the partition) associated with the design.

If the design \( (n, Q) \) has the property that for all possible combinations of \( k \) factors taken \( t \) at a time, \( C(k, t) = \binom{k}{t} \), all \( \prod_{i=1}^{t} q_i \) settings occur, then such a design is called \( t \)-factor covering. Thus, if we define the coverage associated with a design for a given value of \( t \) as

\[
\text{Coverage} = \frac{\text{Number of } t \text{-factor settings covered}}{\text{Total number of } t \text{-factor settings}}
\]

then a \( t \)-factor covering design provides 100% \( t \)-factor coverage. In addition, given a design that is \( t \)-factor covering, we can also investigate the extent to which it covers all combinations of \( t+1, t+2, ..., k \) factors.

**Example of \( t=1 \)-Factor Covering Design**

Table 1 presents an example of a \( t=1 \)-factor covering design \([7]\) applied to 7 factors, where each factor takes on 2 distinct levels. Note that a full factorial design would require \( 2^7 = 128 \) cells of the resulting partition (equal to 128 factor combinations). If we assume one input selected at random for each cell, this implies 128 different inputs. In contrast, the \( t=1 \)-factor covering design given in Table 1 requires only 2 cells of the partition (i.e., only 2 factor combinations) and hence only 2 inputs for software testing. Note that although this design represents a considerable savings over a full factorial design, this design will not always detect faults attributed to 2-way and higher factor interactions.

<table>
<thead>
<tr>
<th>Run</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**Example of Balanced \( t=2 \)-Factor Covering Design**

Consider again the case of 7 factors, where each factor takes on 2 distinct levels. Table 2 presents an example of a \( t=2 \)-factor covering design based on an orthogonal array of strength 2. Orthogonal arrays \([10, 13]\) are balanced fractions of complete factorial designs. (In classical terms, they are main-effect plans with proportional balance.) Orthogonal arrays of strength 2 have been suggested as being especially applicable for software testing \([1]\) because of their ability to uncover faults attributed to 2-way factor interactions. Thus, testing with these designs will uncover faults attributed to two-way factor interactions as well as any higher-level interactions that happen to be present. Note that in Table 2, all possible pair-wise combinations of factor settings are presented in a balanced way (exactly twice in this example). This particular orthogonal array design of strength 2 requires 8 inputs as opposed to the 2 inputs in Table 1 (again assuming a constant allocation scheme of 1 input per partition cell or factor combination).
Table 2: Non-Exhaustive Partition Testing
Test All Combinations Of Factor Pairs
Using An Orthogonal Array of Strength 2

<table>
<thead>
<tr>
<th>Run</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Example of Unbalanced t=2-Factor Covering Design
Consider the case of 10 factors, where each factor takes on 2 distinct levels. Table 3 presents an example of a t=2-factor covering design in which two-way factor combinations are not constrained to be represented an equal number of times. Notice that even though 3 more factors have been incorporated into the design, by removing the requirement for balance, all pairs of factor combinations are now covered by 6 runs so that 100% t=2-factor coverage is achieved with only 6 inputs.

In contrast, orthogonal array designs would require at least 12 test cases or inputs. The savings become even more dramatic when the number of factors is large. For example, with 126 two-level factors, a 10-run design can provide for 100% 2-factor coverage whereas an orthogonal array of strength 2 would require at least 128 runs. This is because the number of runs grows linearly with the number of factors for orthogonal arrays, but only at a logarithmic rate for t=2-factor covering designs [4].

Table 3: Non-Exhaustive Partition Testing
Test All Combinations Of Factor Pairs
Using An “Unbalanced” t=2-Factor Covering Design

<table>
<thead>
<tr>
<th>Run</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
<th>F9</th>
<th>F10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

CASE STUDY
Although t-factor covering designs guarantee coverage of a SUT’s Input domain in the sense that they test all t-wise combinations of test factors, the fault detection capability of these designs has not been established. Nair et al. [9] report that when applying an experimental design approach to test a complex field within a screen, an orthogonal array of strength 2 (16,4^2x2^2) provided the same failure detection probability as a full factorial design. (Nair et al. [9] used a root cause analysis of the relationship between faults and inputs to determine the proportion of failing inputs attributable to a given fault in each of the 64 cells of the partition.) An orthogonal array with run size of 16 had the same probability of uncovering a fault (equal to 1.0) as a full partition test of 64 runs. They also report that a t=1-factor covering design with 4 runs (i.e., a (4,4^2x2^2) experimental design) had a failure detection probability of 0.825. However, the generalizability of their results to other values of t and to other SUTs needs to be investigated.

In this paper, for a given SUT, we investigate the extent to which the software coverage achieved by t-factor covering designs is representative of that achieved by “complete” designs, under different values of t. Our rationale is that if we can provide empirical evidence that for lower order values of t, the software coverage provided by t-factor covering designs is comparable to that achieved by full designs, then we could be more confident of the applicability of lower order t-factor covering designs to software testing. Thus, our empirical work would lend credence to the claim that incomplete experimental designs provide increased test efficiency by reducing the number of partition cells to be tested while, at the same time, providing the same degree of software coverage as exhaustive partition testing.

Instead of fault detectability, we used code coverage as a surrogate measure. Our rationale was as follows:

- First, the software engineering literature indicates that code coverage is correlated with fault detectability [8, 16].
- Second, a set of tools for monitoring code coverage during test execution was readily available to us. Thus, by instrumenting our code, we could easily collect code coverage data.

SOFTWARE UNIT UNDER TEST
Feature Overview
The software system is an operations support system (OS) developed at the former AT&T Bell Laboratories to support maintenance operations to the AT&T network. Thus, any time an event occurred in the network triggering a work item, the on site work force (a technician at a Network Control Center) would create a ticket and use the ticket referral capability of the OS to send the work item to the appropriate work location. The target OS was to execute on a NCR 3600 SMP hardware platform running UNIX® System V R4.0 (NCR Release 2.02). The application was written predominantly in C++ 2.1 using the L4 Standard Components Library.

The software adhered to a client-server transaction-based architecture. Application software executed on a front-end processor while the ORACLE® 6.0.36 Relational Database Management System executed on a back-end processor.
The specific Software Unit under Test executed as two cooperating processes on the front-end processor:

- **Presentation Manager (PM)** provided a character-based user interface that started when a user logged in. PM translated the user's request into service requests and subsequently displayed the resulting response.

- **Database Server** translated database service requests into calls to ORACLE library routines to perform the requested function. Since a server could support several services, the server would first call the correct service function to handle the request. The service function would process the service request and then call the Data Manager class library to interact with the ORACLE database. The service would then return the result to the Presentation Manager.

The specific feature analyzed, **Jeopardy Alerting QUERY**, identifies work items that are in jeopardy of being completed (and hence might not be completed by the specified date). The Jeopardy Alerting QUERY service, QRYJEPACITY (implemented in five C/C++ functions), is a service provided by the Jeopardy Alerting database server that returns jeopardy condition records to a client. This service accepts a query criterion composed of three components:

- **tkt_key**, the identifier of a ticket that can have jeopardy conditions,
- **acty_key**, the identifier for that portion of the ticket that represents work specific to a given office,
- **ofc_id**, the identifier of an office that is responsible for a particular piece of work to be performed on a ticket.

The service then applies this criterion to match jeopardy records stored in a database. One component, **tkt_key**, is mandatory, while the two remaining components, **acty_key** and **ofc_id**, are optional. An additional parameter, **cntl**, determines whether the service should return jeopardy-related information vs. only thereof, and whether the service should apply the query criterion to all jeopardies or only to unresolved jeopardies. If no records are retrieved from the database, the service returns a message indicating that no records were found.

**Feature Partitioning**

For the purpose of comparison, two different partitions were applied to QRYJEPACITY. Partition 1 contained 4 test factors. The product, \(4 \times 3 \times 3 \times 2\), of these factors' levels, resulted in a partition containing 72 cells or unique factor combinations. Partition 2 contained 5 test factors. The product, \(3 \times 3 \times 3 \times 3 \times 2\), of these factors' levels, defined a partition containing 162 cells or factor combinations.

**Code Coverage Measures**

A SUT can be represented as a directed graph composed of a set of nodes and a set of edges where:

- A node is defined as a block of consecutive statements or expressions such that if one element of a block is executed, then all elements of the block are executed.
- An edge is defined as a control flow branch from the last element of the source block to the first element of the destination block.
- A (complete) path is defined as a sequence of blocks starting with a Start block and ending with a Stop block in which a branch connects block \(_i\) to block \(_i+1\) except for block \(_1\). The jeopardy Aleiling QUERK service accepts a query criterion composed of three components:
  - **ids**, the identifier of an office that is responsible for a particular piece of work to be performed on a ticket.
  - **tkt_key**, the identifier of a ticket that can have jeopardy conditions,
  - **acty_key**, the identifier for that portion of the ticket that represents work specific to a given office,
  - **ofc_id**, the identifier of an office that is responsible for a particular piece of work to be performed on a ticket.

An example of such a graph representation is given in Figure 2, which presents a program graph for a simple program that reads an arbitrary length sequence of numbers and prints each number's absolute value. In that example, a sequence of exclusively negative numbers would cause every block to be visited, but the branch from block 3 to block 5 would not be traversed.

The adimion family of tools [5] collects statistics on both block and path coverage. Consequently, our two code coverage measures for a given test case execution were a) the number of unique blocks executed and b) the number of unique paths executed.

**RESULTS**

**Random Experimental Designs**

The first set of analyses analyzed the \(t\)-factor coverage measure given in Equation 2. Specifically, it addressed a) how \(t\)-factor coverage varies with size of the experimental design, b) the interrelationship between \(t\)-factor coverage for different \(t\), and c) how code coverage varies as a function of \(t\)-factor coverage. These analyses employed a common methodology as follows:

- for \((partition = 1, 2)\)
  - for \((n = 1, 2, ..., Q)\)
    - for \((n, Q)\)
      - for \((ides = 1, 2, ..., 10)\)

The addmon family of tools [5] collects statistics on both block and path coverage. Consequently, our two code coverage measures for a given test case execution were a) the number of unique blocks executed and b) the number of unique paths executed.
Due to space limitations, only the results from partition 2 (the finer partition) are reported since the partition 1 results are consistent with findings observed for partition 2.

**t-Factor Coverage vs. Size**
Figure 3 shows how t-factor coverage varies as a function of experimental design size for \( t = 1, 2, \ldots, k \). For \( t = k \), note that t-factor coverage increases linearly with experimental design size (which is the result of random sampling without replacement) and that all cells of the partition (i.e., a full factorial design) must be presented to attain 100% t-factor coverage. As the value of \( t \) decreases, note that it becomes increasingly easier to obtain 100% t-factor coverage with a smaller sized experimental design. Thus, for the case of \( t = 1 \), we can obtain 100% t-factor coverage with a relatively small sized experimental design.

**Inter-Relationship Between t-Factor Coverage Values**
Figure 4 presents a scatterplot matrix of the \( k = 5 \) t-factor coverage measures for all possible pairs of factor coverage measures. Except for \( t = 1 \)-factor coverage (first row and first column of the matrix), Figure 4 indicates that the t-factor coverage measures exhibit nice (i.e., "tight") functional interrelationships.

For partitions having the same number of levels for all factors (i.e., for \( q^k \) designs), there is a simple formula for the expected t-factor coverage of a random design as a function of the shape of the design:

\[
E(Coverage_t) = 1 - (1 - 1/q^t)^n
\]  

(3)

Furthermore, when \( n \) is large, there is a nearly functional relation between t-factor coverage and \( t' \)-factor coverage for any \( t, t' \):

\[
\log(\frac{1 - Coverage_t}{1 - Coverage_{t'}}) = \log(\frac{1 - 1/q^t}{1 - 1/q^{t'}})
\]

(4)

**Code Coverage as a Function of t-Factor Coverage**
Figure 5 right column presents block coverage as a function of t-factor coverage for different values of \( t \). Block coverage as a function of experimental design size has also been included to emphasize the equivalence between \( t = k \)-factor coverage and experimental design size.

For \( t = k = 5 \), Figure 5 right column indicates that block coverage increases rapidly to an asymptotic value and then remains at this value for the duration of the testing. Assume that a tester had an objective of attaining 100% \( t = k \)-factor coverage. Clearly, this is too stringent an objective since the maximum number of blocks executed (19 blocks) peaks way before this value—at a \( t = k = 5 \)-factor coverage value of 0.40. Further testing beyond a \( t = 5 \)-factor coverage of 0.40 (or beyond 65 test cases, 1 test case per partition cell) is clearly wasteful in terms of attaining additional block coverage.

In contrast, assume a tester has the objective of attaining 100% \( t = 1 \)-factor coverage. The top plot in Figure 5 right column indicates that this objective is too lenient since there is wide variation in the number of blocks (13.7-19) executed at \( t = 1 \)-factor coverage of 100%. At 100% \( t = 2 \) or \( 3 \)-factor coverage, however, the number of blocks executed is maximum and, in addition, exhibits little variation (especially for \( t = 3 \)-factor coverage). Consequently, attaining 100% t-factor coverage for a lower order value of \( t = 2 \) or 3 will ensure that the maximum number of blocks will be consistently executed with the fewest number of test cases.
Figure 5 left column presents path coverage as a function of $t$-factor coverage for different values of $t$. Note that a higher value of $t$ ($t=4$) is required to approach the maximum number of paths executed.

**Systematic Experimental Designs**

Three systematic test designs were constructed to obtain 100% $t=2$, $t=3$, and $t=4$-factor coverage with the minimal number of runs. (These designs are called $t=2$-factor covering, $t=3$-factor covering, and $t=4$-factor covering designs, respectively.) These designs are efficient designs in the sense that they achieve a higher $t=2$- or $t=3$- or $t=4$-factor coverage when compared to random experimental designs of the same size.\(^4\)

**Comparison of Code Coverage Achieved by Systematic vs. Random Designs**

Figure 6 right column compares the block coverage obtained by systematic vs. random designs in terms of boxplot displays. For the systematic $t=2$-factor covering design, the median number of blocks executed is slightly higher than for a random experimental design of the same size (17 vs. 16 blocks) while for the systematic $t=3$- and $t=4$-factor covering designs, the median number of blocks executed is the same (19) as a random experimental design of the same size. Note that the median $t=2$-factor coverage value for random experimental designs of size 10 is 0.76 and that the median $t=3$-factor coverage value for random experimental designs of size 30 is 0.79. Furthermore, since the maximum number of blocks that could be executed was 19, the finding that systematic $t=2$- and $t=3$-factor covering experimental designs did not achieve much higher block coverage than random experimental designs is not surprising.

For $t=2$-factor and $t=3$-factor covering systematic designs, note that the variation in number of blocks executed over different instantiations of a design is smaller for the ease of the systematic design than for the random design. Consequently, these systematic designs are more reliable in achieving block coverage than random experimental designs of the same size. For the $t=4$-factor covering design, the systematic design achieves almost exactly the same distribution of number of blocks executed (19 blocks) as a random experimental design of the same size. This is consistent with Figure 5 right column which indicates that block execution peaks prior to 100% $t=4$-factor coverage.

Figure 6 left column provides a similar comparison between systematic and random designs with respect to path coverage. Figure 6 left column indicates that the systematic $t=2$-factor covering design exhibits the same median number of paths executed (4) as random designs of the same size but that the inter-quartile range is smaller for the systematic design. The systematic $t=3$-factor covering and $t=4$-factor covering designs have a median number of paths executed that is slightly higher than random experimental designs of the same size and again, the inter-quartile range is smaller for the systematic design. Thus, these results are consistent with Figure 5 left column which indicate that for path coverage, a higher-order value of $t > 2$ or $t=4$, is required to approximate path coverage attained by exhaustive partition testing.

**CONCLUSIONS AND IMPLICATIONS**

In this paper, we have described an approach to functional testing that can be applied early in the software development cycle, even before code is developed. In this approach, feature requirements are analyzed with respect to factors and levels and then these factors are used to partition the set of potential inputs into cells or subgroups. Inputs are then selected by applying a $t$-factor covering design to select a restricted number of subgroups (where the covering design guarantees, a priori, the degree of functional coverage—pairwise, triplets, quadruplets, etc.) and then sampling inputs within each selected subgroup. Such an approach would be useful when knowledge of the software development process can be used to select factors whose settings are related to failure occurrence and when the number of partition cells is large enough to preclude exhaustive partition testing. Our empirical work suggests that testing from all cells of the partition may be redundant in terms of code coverage and that lower-order $t$-factor covering designs may be effectively used to reduce this redundancy at least for certain types of code coverage (i.e., block coverage). Clearly, more empirical work is required with more “substantial” SUTs (i.e., SUTs with more factors) and with other measures of software and code coverage.

**Constructing Covering Designs**

Several technical challenges remain in applying experimental designs to software testing. First is the issue of constructing covering designs. Sherwood [12] describes the Constrained Array Test System (CATS) that generates covering designs using a greedy search algorithm (i.e., a procedure in which, at each stage, it looks only one step ahead as opposed to taking a long-term global approach). The user can start with a set of runs that take into account the importance of the test factors so that the more critical settings will be covered initially. Given $Q$ and $t$, a list of feasible runs is then generated where a feasible run is defined as a run that is not specifically prohibited from

---

\(^4\) The systematic design with 10 runs obtained 100% $t=2$-factor coverage vs. a median $t=2$-factor coverage of 0.76 obtained by random experimental designs of the same size. The systematic design with 30 runs obtained 100% $t=3$-factor coverage vs. a median $t=3$-factor coverage of 0.79 obtained by random experimental designs of the same size. The systematic design with 81 runs obtained 100% $t=4$-factor coverage vs. a median $t=4$-factor coverage of 0.83 obtained by random experimental designs of the same size.
being included in the list. The next run is selected (from those remaining) on this “possibles” list. CATS goes through the list and works out for each prospective run, how many new factor combinations are covered. This results in a smaller list containing just those runs that achieve the maximum number of new covers. CATS then simply selects the first one in this sublist.

To handle large Q where an exhaustive search of all possibilities is not possible, the algorithm chooses a subset, \( f_1 \), of factors, and a covering design, \( D_1 \) for these factors is generated. Then, an additional subset of \( f_2 \) factors is chosen and a list of feasible runs is generated by concatenating the runs of \( D_1 \) with all possible settings of these new factors. A new covering design \( D_2 \) for the \( f_1 + f_2 \) factors is generated by selecting from the list. This procedure is iterated until all the factors have been entered. This strategy ensures that the size of the list of runs to be considered remains manageable. Dalal and Mallows [4] indicate that other algorithms are possible in cases where \( Q = q^k \) by exploiting the symmetry of the problem and by randomizing the selections at each stage.

Cohen and others [3] describe the Automatic Efficient Test Generator system, AETG, which appears to have similar functionality as CATS but uses more efficient algorithms.

**Role of Random Experimental Designs in Software Testing**

A second issue in applying experimental designs to software testing has to do with using random experimental designs.

First, assume we assign some value of \( t \). We then select a fraction, \( f \), in (0,1) to obtain an \( f \times 100\% \) \( t \)-factor coverage objective. For a random experimental design, we can ask what value of \( n \) is required to attain our \( f \times 100\% \) \( t \)-factor coverage objective. For the case where \( Q = q^k \), let \((n, q^k, t, f)\) denote an experimental design that covers a fraction, \( f \), of \( \left[ \frac{k}{t} \right] q^t \) possible \( t \)-factor combinations and that therefore will meet our \( f \times 100\% \) \( t \)-factor coverage objective. Roux [11] presents the result that for any \( f \) in (0,1), there is an \( n \) depending on \( q, t \) and \( f \) but not on \( k \) such that for every \( k \), no matter how large, an \((n, q^k, t, f)\) design exists. Consequently, for the class of random designs where \( Q = q^k \), one can always find a random experimental design of size \( n \) that will meet a \( f \times t \)-factor coverage objective. This argument also applies for general \( Q = \prod_{i=1}^{k} q_i \), provided \( q_i \leq q \) for all \( i \).

Second, given an experimental design \((n, q^k, t, f)\) that satisfies our \( f \times t \)-factor coverage objective, one can also determine what \( t \)-factor coverage it will achieve for some \( t' \neq t \). Mallows [6] shows that for large \( n \), it depends (asymptotically) only on \( t \) and \( f \) and not on \( n \) or \( k \) (see also Equation 4). E.g., for \( q = 2 \), if a large random design achieves 99\% \( t = 2 \)-factor coverage, then this will also achieve 40\% \( t = 5 \)-factor coverage. Since these issues—random design size required to satisfy an \( f \times t \)-factor coverage objective, and \( t \)-factor coverage given \( f \times t \)-factor coverage—are relevant to software testing, extension of Mallows’ asymptotic results would seem desirable.

In summary, although the application of experimental designs to software testing may appear to be similar to the classical application of experimental designs for estimating main and interaction effects, a different class of experimental designs appears to be required for software verification. Hence, a different class of issues must be resolved. More research is required, both in software engineering and in mathematical statistics.

**ACKNOWLEDGEMENTS**

We would like to thank Debbie Brown for the establishment of the QRYJEPACTY test database; Diane Cameron for SABLIME support of the QRYJEPACTY service; and Elaine Weyuker for discussions on code coverage.

**REFERENCES**


---

**Figure 3.** t-factor coverage vs. size (Partition 2).
Figure 4. Pairwise correlations between t-factor coverage levels (Partition 2).
Figure 5. Path coverage vs. block coverage as a function of t-factor coverage (Partition 2).
Figure 6. Comparison of systematic vs. random designs with respect to path and block coverage (Partition 2).