

Factor-Covering Designs for Testing Software*

Siddhartha R. Dalal

Bellcore

Morristown, NJ 07960

(sdalal@notes.cc.bellcore.com)

Colin L. Mallows

AT&T Labs–Research

Florham Park, NJ 07932-0971

(clm@research.att.com)

Testing is a critical component of modern software development. The problem of designing a suite of test cases is superficially similar to that of designing an experiment to estimate main effects and interactions, but there are crucial differences. Additive models are unhelpful, and classical design criteria are also. We propose a new class of models, and new measures of effectiveness. We compare several designs.

KEY WORDS: Equivalence partitioning; Orthogonal arrays; Random balance; Supersaturated designs.

Development and production of high-quality software at a reasonable cost is a critical issue for today's products and services. Software testing is used to validate the correctness of programs as well as to weed out defects, and is typically the most expensive phase of the production process. Thus the problem of performing effective and economical testing is a key issue. Papers on this subject appear in proceedings of several major conferences on testing (prominent ones in the US are International Conference on Testing, STAR conference, ICSE, and ISSRE), and in various IEEE and ACM journals.

In the statistics literature the issue of software testing has been discussed in several recent papers. Most of these discuss modeling reliability, and the problem of when to stop testing (see, e.g. Dalal and Mallows, 1988, Singpurwalla, 1991, Dalal and McIntosh, 1994, and references therein). In these works the process of testing is assumed to have an unknown but fixed productivity, and the emphasis is on the estimation and prediction of reliability. Here we address the issue of improving productivity.

In testing we must generate test cases, administer them,

correct faults when they are found, and manage the whole enterprise. Thus, there are (at least) three ways productivity can be improved, namely, generation of more efficient sets of test cases, automation of the testing process, and improvement of the management process. The last two issues are hard to influence since they require organizational changes and commitments (for process improvement paradigms, see Humphrey (1989)). Here we concentrate on the first issue, that of designing test cases.

The problem of designing a batch of tests for a large software product is superficially similar to that of designing an experiment for estimating main effects and interactions. In fact classical designs have been proposed for this purpose (Mandl (1985), Tatsumi, Watanabe, Takeuchi and Shimokawa (1987), Zeitler (1991), Brownlie, Prowse and Phadke (1992)). However in software testing the usual additive models do not apply since there is no concept of estimation of main effects (see Sections 3 and 4 below), and a different class of designs becomes attractive. The purpose of this paper is to describe this approach, to survey known results and present a few new ones, and to suggest some research opportunities.

In Section 1 we survey of some of the existing methodology used in software testing, giving several examples.

*Appeared in *Technometrics* 40(3), August 1998, pp. 234–243. Copyright © 1998 American Statistical Association and the American Society for Quality.

In Section 2 we show the need for a new class of experimental designs, which we call factor-covering designs. In Section 3 we describe a new model for software faults. Section 4 presents our notation for factor-covering designs, and defines two new indices of merit for them. In Section 5 we review what is known about their construction. In Section 6 we present some numerical comparisons (using our indices) involving various classes of designs. In Section 8 we briefly discuss two automated tools for generating factor-covering designs, and in Section 8 we describe a telecommunications application. In Section 9 we discuss evidence of effectiveness of this approach. Finally, in Section 10 we list several open issues.

1 Software Testing

Testing is a critical activity since a single fault can cripple a whole system and result in great loss, as seen by problems with Intel's Pentium chips, breakdowns of the telecommunications systems in major US cities, etc. Testing is expensive. It often consumes between 1/3 to 1/2 of the total cost of software development. Even larger numbers are typical for safety-critical systems such as nuclear power plants and defense systems. Another example is the Year 2000 problem. For various reasons, including shortage of memory, it was customary to use a 2 digit year rather than a 4 digit year. This creates a problem, since 00 can refer to either 1900 or 2000. Further, 2000 will be a leap year while 1900 was not. The total cost worldwide of changing and testing the relevant software is estimated to be 400 billion dollars (See *The Economist*, 1997). It is expected that about 50% of the cost will be in testing the software.

The process of software testing is typically divided into various phases: Unit testing (testing of small pieces of code written typically by one programmer), Integration testing (testing of several subsystems, each of which is comprised of many units) and System testing (testing of combination of subsystems). There may be still further phases such as Acceptance testing (when the software is first delivered to a client), Alpha testing (unofficial trials by willing customers before full productization), FOA (First Office Application), etc.

Besides these stages of testing, there are many different methods of testing. Structural testing or, White box

testing, refers to the type of testing in which tests are designed on the basis of detailed architectural knowledge of the software under test. Functional testing, or Black Box testing, (Jorgensen, 1995) refers to the type of testing where only the knowledge of the functionality of the software is used for testing; knowledge of the detailed architectural structure, or of the procedures used in coding, is not utilized. Structural testing is typically used during unit testing, where the tester (usually the developer who created the code) knows the internal structure and tries to exercise it based on detailed knowledge of the code. Functional testing is used during integration and system test, where the emphasis is on the user's perspective and not on the internal workings of the software. Functional testing tries to test the functionality of the software as it is perceived by the end users (based on user manuals) and the requirements writers. Thus, functional testing consists of subjecting the system under test to various user controlled inputs, and watching its performance and behavior. The primary focus of this paper is on functional testing. Beizer (1995) provides an excellent introduction to this subject and talks about various objectives of testing in this context.

Since the number of possible inputs is typically very large, testers need to select a sample, commonly called a *suite*, of test cases, based on effectiveness and adequacy. Much functional testing is done in an intuitive and less formal manner. Typically, testers, working from their knowledge of the system under test and of the prospective users, decide on a set of specific inputs. Clearly there is the possibility that important interactions among the inputs will be missed. Herein lie significant opportunities for a systematic approach, based on ideas from sampling and experimental design theory. Consider the following example.

Example: Testing an Air to Ground Missile System. Consider a software system controlling the state of an air to ground missile. The key inputs for the software are the altitude, attack and bank angles, speed, pitch, roll, and yaw. (There are many more- e.g. ambient temperature, pressure, wind velocity, etc., We will consider these later on.) Typically, these variables do not have any joint constraints as far as the software is concerned.

To test this software system, combinations of all these inputs must be provided and the output from the software system checked against the corresponding physics. Each

combination tested is called a test case. One would like to generate test cases that include inputs over a broad range of permissible values.

Since in this example, we have continuous variables as inputs, the total number of possible test cases is unlimited. To reduce the number of test cases, testers have developed a number of heuristic strategies. Two guiding principles are i) non-redundancy (test cases are chosen so that for each test case that is included, the number of test cases which remain to be tried is reduced by at least one), and ii) generality (the outcome of the test case is generalizable beyond the specific inputs used). To implement these principles a number of concepts have been developed which can be applied in conjunction with each other.

One of these relates to the notion of Equivalence Partitioning (Myers 1979). It is assumed that the range of each of the input variables can be divided into a number of mutually exclusive classes, called equivalence classes, with the property, that the outcome of a test is generalizable to the entire equivalence class. That is, the same outcome would be expected regardless of a specific input value from that class. Myers states : “one can reasonably assume that a test of a representative value of each class is equivalent to a test of any other value.” Since one cannot “reasonably assume” unless there is only one member in an equivalence class, in practice testers divide the input domain into a number of possibly overlapping classes (but usually with very little overlap) and select from 1 to 3 distinct inputs as representatives from each class. Typically there is much freedom in choosing the partitioning.

Having formed the equivalence partitioning, one still needs to decide which members be considered as representative members. This is where another notion, that of Boundary Value Analysis is applied. This is based on the experience that “test cases that explore boundary conditions have a higher payoff than test cases that do not” (Myers, 1979). Boundary conditions are described as “those situations directly on, or above, and beneath the edges of input equivalence classes...”. Thus, this concept is similar to that of a minimax strategy. Let us illustrate these concepts in context of the example we discussed above.

Example (continued): Testing an Air to Ground Missile System. To determine the test cases, we can go back to the requirement document. Suppose we are interested

in testing the response during attack maneuvering. We will know the maximum and minimum possible values for each variable. Thus, we could choose to have a set of equivalence classes, each corresponding to the range of one variable. The concept is similar to that of a factor in standard design theory. Now we have the problem of selecting representative values. Following the boundary value heuristic, we will select the maximum and minimum as the two representative values for each of the 7 input variables. Since normal maneuvering lies within these limits, we may want to include one or more intermediate values, for example a mid-point. Let the lower, middle, and upper values be input states 1,2,3 respectively. Then in the language of statistical experimental design, we have seven factors, A,..., G (altitude, attack angle, bank angle, speed, pitch, roll, and yaw), each at three levels. To test all the possible combinations, one would need a complete factorial experiment, which would have $3^7 = 2187$ test cases consisting of all possible sequences of 1's 2's and 3's. With only two levels per factor, we need $2^7 = 128$ test cases.

The notions of equivalence partitioning and boundary value analysis have achieved a tremendous reduction from the effectively infinite problem we would face in exhaustive testing. In this example, it may be possible to test all 128 or 2187 test cases, but, this may be only one of many systems that need to be tested. Further, if we want to increase the scope of the testing by including three more variables (for example H: ambient temperature, I: pressure, and J: wind velocity) then we would have $2^{10} = 1024$ or even $3^{10} = 59049$ test cases. In anything other than toy problems, the situation is typically much worse than this. Many of the dimensions are categorical, and thus, there is no reduction feasible due to boundary value analysis, and even using automated testing, the number of cases is impossibly large. We illustrate this by giving two more examples.

Example: Screen Testing. Typically, users of business systems interact with software via a succession of screens, each of which has a number of fields. It is not uncommon to have 50 or more fields, for example Cohen, Dalal, Kajla, and Patton (1994) give an example of a screen with 76 fields. Assuming only 2 values per field, (for example “valid” and “missing”), one has 2^{76} test cases. At a rate of a million cases per second (impossible to achieve today even with automation), this would require 2.10^{15} years to

test.

Example: Interoperability Testing. Periodically, software companies update their products, and sell them as new versions (e.g. Windows 95 vs. Windows 3.1). When these products come out, it is essential that they work with a number of versions of other software products. Thus, the issue of interoperability testing is critical. For example, one would want Windows95 and 3.1, Word 6.0 and 2.0, Excel 5.0 and 4.0, etc, all to work with one another. Thus, suppose one had four software products, and wanted to support two versions of each, then we must study 28 interoperability problems. However each interoperability problem represents a large number of sub-problems, and detailed analysis may result in a huge number of factors to be studied. Also in each version of the software, there may be a large number of parameter-settings, which may have unpredictable influences on the results of a test. In such a case we may choose to regard the test environment as being random, so that the outcome of a test is not deterministic, but will identify a fault only with a certain probability that is strictly between 0 and 1.

There are many other examples showing the geometric explosion in the number of test cases- see Dalal and Patton (1993) for feature interactions testing and Burroughs, Jain and Erickson (1994), for protocol testing. Sloane (1993) gives references to applications to hardware testing including circuit testing, network testing, etc.

2 Covering Designs

In a statistical experiment if one is interested in only main effects, then one can get away with a highly fractionated factorial design. However in the case of software testing, there is no interest in estimating additive effects. Interest lies in covering the test space as completely as possible and checking whether the test cases pass or fail. However, it is certainly possible to use standard statistical designs. For example consider the set of test cases given in Table 1.

This is an orthogonal array of strength two, with 7 factors and 2 levels. It requires 8 test cases instead of 128. It is clear that all possible pairwise combinations of levels of two factors are covered in a balanced way (exactly twice each in this example). Thus testing according to this design will protect against any incorrect implementation of the code involving any pairwise interaction, and

Table 1: An orthogonal array

Test	Factor						
	A	B	C	D	E	F	G
1	1	1	1	1	1	1	1
2	1	1	1	2	2	2	2
3	1	2	2	1	1	2	2
4	1	2	2	2	2	1	1
5	2	1	2	1	2	1	2
6	2	1	2	2	1	2	1
7	2	2	1	1	2	2	1
8	2	2	1	2	1	1	2

also whatever other higher order interactions happen to be represented in the table. Brownlie, Prowse and Phadke (1992) have suggested the use of orthogonal array designs of strength two for testing.

However, in software testing repeating a run with exactly the same inputs will give exactly the same output, so exact replication is unhelpful, and wasteful. The problem is not that of estimating an unknown response function, but rather that of determining whether the software functions correctly under all relevant input conditions; the response is either “O.K.,” in which case nothing needs to be done, or “failure” in which case the test case can be analyzed to determine the cause (or causes) of the failure. The efficiency of a test design is measured by the degree to which it covers the relevant input space. If we insist on coverage of all pairs, but give up the restriction of balance, we can do a lot better. For example, Table 2 gives a design we call R_6 , which achieves 2-coverage of 10 two-level factors, in just 6 test cases.

Table 2: The design R_6

Test	Factor									
	A	B	C	D	E	F	G	H	I	J
1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	2	2	2	2	2	2
3	1	2	2	2	1	1	1	2	2	2
4	2	1	2	2	1	2	2	1	1	2
5	2	2	1	2	2	1	2	1	2	1
6	2	2	2	1	2	2	1	2	1	1

The structure of this design is clear: the first test case sets all factors to level 1; in the last five runs, all combinations of two 1's and three 2's are used, once each. Then in each pair of columns, we see (1,1) in the first row, and (2,2) at least once in the last five rows (since the sets of three 2's must overlap); and since the columns are not identical and each contains three 1's and three 2's, both (1,2) and (2,1) must occur also. Thus in every pair of columns, all four possibilities (1,1),(1,2),(2,1),(2,2) each occur at least once. All pairs of values are covered in only 6 test cases.

Thus, in the missile example, besides incorporating the 7 factors already mentioned, one can now include three more factors and still guarantee protection against any pairwise interaction, using only 6 test cases. Unlike orthogonal arrays, for which the number of cases grows at least linearly with the number of factors, covering designs grow at only a logarithmic rate. The savings are thus more dramatic when the number of factors is large; for example with 126 binary factors we can cover all pairwise interactions with only 10 runs, whereas an orthogonal array would require at least 128 runs.

Clearly two-level factor-covering designs could be used for the “sparse effects” problem, where “supersaturated” designs have been proposed by Booth and Cox (1962), Lin (1993,1995), and Wu (1993). We have not seen any discussion of sparse effect problems with factors having more than two levels. In Table 3 we give two 2-covering designs for factors with three levels.

3 A Model for Software Faults

Suppose we have defined k factors, with the i -th having q_i levels. A t -factor fault is one which is triggered whenever some set of t factors is held at some set of levels. Thus there are $\sum_i q_i$ possible 1-factor faults, $(1/2) \sum_{i \neq j} q_i q_j$ possible 2-factor faults, ... $\prod_i q_i$ possible k -factor faults. Each possible fault can be specified by giving the relevant factors and levels; we use the notation $f = (f_1, f_2, \dots, f_k)$ where $0 \leq f_i \leq q_i$ and $f_i = 0$ means that the i -th factor setting is irrelevant. Thus for example if we have six factors, each with 3 levels, a possible fault is $f = (2, 3, 0, 0, 1, 0)$, so that three of the factors are irrelevant but the other three must be set at particular levels to trigger the fault. Let F_t be the set of all such fault-vectors

Table 3: Two 2-Covering Designs

A ₁₅			A ₁₈			
1111	1111	1111	1111	1111	1111	1111
1222	1222	1222	1222	1222	1222	1222
1333	1333	1333	1333	1333	1333	1333
2123	2123	2123	2123	2123	2123	2123
2231	2231	2231	2231	2231	2231	2231
2312	2312	2312	2312	2312	2312	2312
3132	3132	3132	3132	3132	3132	3132
3213	3213	3213	3213	3213	3213	3213
3321	3321	3321	3321	3321	3321	3321
1111	2222	3333	1111	1111	1111	1111
2222	3333	1111	1111	2222	2222	2222
3333	1111	2222	1111	3333	3333	3333
1111	3333	2222	2222	1111	2222	3333
2222	1111	3333	2222	2222	3333	1111
3333	2222	1111	2222	3333	1111	2222
			3333	1111	3333	2222
			3333	2222	1111	3333
			3333	3333	2222	1111

having exactly t non-zero elements, i.e. corresponding to t -factor faults, and let F be the union $\cup_{t=1}^k F_t$. The number of elements in F is $\prod_i (1 + q_i)$.

This notation is not completely satisfactory. For example, if $k = 4$ and $q_1 = q_2 = 2$, the 1-factor fault (1, 0, 0, 0) can be regarded as a pair of 2-factor faults, (1, 1, 0, 0), and (1, 2, 0, 0). This might be entirely appropriate, if there are two distinct errors in the software. Clearly the 1-factor description is more parsimonious, but in general parsimony is not uniquely defined; for example the triad of 2-factor faults (1, 1, 0), (1, 2, 0), (2, 1, 0) can be expressed as the union of a 1-factor fault and a 2-factor fault (in two distinct ways). We ignore this difficulty since in practice faults occur sparsely and minimal descriptions are usually unique.

Identifying faults using the standard notation for additive models is possible, but very clumsy. For example to identify the single three-factor fault (1,1,1,0) we need three main effects, three 2-factor interactions, and one 3-factor interaction. Also we need a non-linear transformation of the response function into the set {0,1}.

A probability model for the faults in a software mod-

ule must specify the probability that each possible fault is present. Thus a general model would have $|F|$ parameters. Simplifications can be made if we assume symmetry; thus one simple model depends on a set of k parameters p_1, p_2, \dots, p_k , and assumes that each fault in F_t occurs with probability p_t , with different faults occurring independently. Note that in this model, ignoring the multiple-description difficulty alluded to in the previous paragraph, the probability that the i th factor figures in a 1-factor fault is proportional to q_i . An alternative model would make these probabilities all equal. In the following, we will assume that all the p 's are small, so that at most one fault appears. Also we will assume that at most one p is non-zero.

In the absence of *a priori* knowledge, a symmetry assumption seems reasonable. However, if we do not have a stochastic environment, the independence aspect of this model is not empirically verifiable, since it is impossible in principle to obtain an empirical estimate of the probability that a particular fault occurs. We can interpret the parameters p_t as being subjective probabilities. If the levels of the various factors have been randomized, the symmetry assumption is natural. If we accept the independence assumption, the parameter p_t can be estimated straightforwardly (by testing all t -factor settings).

If our problem does involve a stochastic environment, we may assume that a test case corresponding to a particular fault f will result in a failure only with some non-trivial probability r_f . For simplicity, we may choose to assume that these probabilities are all equal. Mallows (1997) discusses the stochastic-environment problem.

If we use this model, experience with a class of similar systems may lead to useful prior estimates of the parameters. In a later section we report some empirical evidence.

4 Factor-covering Designs— Definitions, Preliminaries and Properties

Definition. Suppose we have n runs and k factors, with the i -th factor having q_i levels. Then the design is called a (n, Q) design, where Q stands for the word $q_1 q_2 \dots q_k$.

Thus the design R_6 displayed in Table 2 is a $(6, 2^{10})$ design. (The reader should be aware that our notation,

while perhaps natural for a statistician, differs from that used in many previous papers on this subject. Also the term “covering design” has been used in other senses from ours.) Such a design is a collection of n row-vectors of length k , with the i -th element of each vector being drawn from the alphabet $\{1, \dots, q_i\}$.

Definition. If an (n, Q) design has the property that the projection onto any t coordinates exhibits all $\prod q_i$ possibilities, we say such a design is t -covering. A t -covering design is *optimal* if n is minimal for fixed Q, t .

Other names that have been used for these designs are t -surjective array, or (for the transposed array) a qualitatively t -independent family of vectors. Box and Tyssedal (1996) call a t -covering $(n, 2^k)$ design a (n, k, t) screen, and investigate the coverage properties of certain orthogonal arrays.

R_6 is a 2-covering design of type $(6, 2^{10})$. In fact it is the optimal 2-covering design. Of course, it is not a 3-covering design; for 20 triads the setting (1,1,1) occurs twice, for another 10 triads (2,2,2) occurs twice, and for another 30 triads (2,2,1) (or some permutation of this) occurs twice; for the remaining 60 triads of factors, all six runs give distinct settings. Thus this design covers only 660 of the 960 possible triad settings.

This example raises some general questions. How well can a t -covering design cover the possible combinations of $t+1, t+2, \dots$ factors? (Mallows (1997) gives an asymptotic answer to this question for random designs.) More generally, how can we measure the efficiency of an (n, Q) design? We propose two indices of merit of an (n, Q) design.

Definition. For each t , the t -coverage of a design is the ratio of the number of t -factor settings that are covered to the total possible number of t -factor settings.

Consider a particular (n, Q) design, and suppose there are $n_t(j)$ t -factor settings that are each covered exactly j times, $j = 0, 1, 2, \dots$. Then the number of different t -factor settings that are covered is $\sum_j n_t(j)$, and the total number of t -factor settings that are exercised by the design (counting multiplicities) is $\sum_j j n_t(j)$. For an (n, q^k) design we thus have

$$t\text{-coverage} = \frac{\sum_j n_t(j)}{q^t C(k, t)}$$

where, $C(k, t)$ is the binomial coefficient “ k choose t ”.

The “ t -coverage” tells what fraction of the possible t -factor settings are covered by the design; 100% is perfect, all possible t -factor settings are covered.

Definition. For each t , the t -diversity of a design is the ratio of the number of t -factor settings that are covered to the total number of t -factor settings that occur in the design.

Thus this index tells to what degree the design avoids replication; 100% says that all t -factor settings that appear in the design are different, so the design does a perfect job in avoiding repetitions; 50% says that each setting that is covered is exercised twice (on the average). This suggests that we might be able to do better, i.e. achieve higher coverage, by working towards higher diversity. If the t -coverage of a (n, q^k) design is c , the t -diversity is cq^t/n , which is a number independent of the detailed structure of the design, and even independent of k .

If two designs have the same t -factor coverage but one has fewer runs than the other, the difference can be measured in two (equivalent) ways; first, simply by the ratio of the numbers of runs, and second by the fact that the smaller design has larger diversity; the larger design has more t -factor settings altogether, but the extra ones are wasted on replicating settings that are already covered. Again, if the t -diversity is 100%, the t -coverage for a q^k design must be n/q^t , and there is no way to get any higher value; this design does a perfect job of spreading its effort. However if the t -diversity is only 50%, there might be another design (of the same size) with higher coverage (and higher diversity). Computation of these coverage and diversity measures is tedious but straightforward; for large designs with much symmetry they pose some interesting combinatorial challenges.

For the R_6 design in Table 2, the 3-coverage is $660/960 = 68.8\%$, and the 3-diversity is $660/720 = 91.7\%$ suggesting that this is close to the best we can do for a 6 run design and that to increase the coverage we will have to increase n . It is believed that we need 12 runs to cover all triads completely.

5 Some Known Designs

In the case $q = t = 2$ the optimal designs are straightforward generalizations of the $(6, 2^{10})$ design given above, as was proved by Renyi (1971) (for n even), and by Ka-

Table 4: Sizes of the Best Known 2-Covering $(n, 3^k)$ Designs

k	2	3	4	5	6	13	16	18	56	126
n	9	9	9	11	12	15	18	21	24	27

tona (1973) and Kleitman and Spencer (1973) (for all n). The design R_n (our notation honors Renyi) uses n runs to cover $k = C(n - 1, \text{ceiling}(n/2))$ factors pairwise ($t = 2$). Thus R_8 covers $C(7, 4) = 35$ two-level factors pairwise, R_9 covers $C(8, 5) = 56$ two-level factors pairwise, and R_{10} covers $C(9, 5) = 126$ two-level factors pairwise.

In the case $q=2, t = 2$ it has been shown by Gargano, Korner, and Vaccaro (1993) that for k large the minimal n satisfies

$$n = \frac{q}{2} \log_2 k(1 + o(1)) \quad (1)$$

and Godbole, Skipper and Sunley (1995) have shown that for all q, t the minimal n satisfies

$$n \leq (t - 1) \log k / \log \left(\frac{q^t}{q^t - 1} \right) (1 + o(1))$$

but no explicit general constructions are known. The smallest known values of n such that a 2-covering design exists for k three-level factors are given in Table 4 (Sloane (1993), $k = 13$ by Cohen and Fredman (1996) and $k = 18$ and $k = 56$ by Seymour (Personal Communication).

Larger designs can be constructed by taking three copies of an R_n , removing the first row from each, and writing the first copy in terms of levels 1,2, the second copy in terms of levels 2,3, and the third copy in terms of levels 3,1. This gives a $(3(n - 1), 3^k)$ design with $k = C(n - 1, \text{ceiling}(n/2))$. Note that this family of 3-level designs is very far from optimal for n large; according to (1) each design uses about twice as many runs as would be needed by an optimal design.

Sloane (1993) surveys the case $q = 2, t = 3$ in detail, and gives the following upper bounds to the minimal number of runs for a 3-covering of k two-level factors:

Thus with 12 runs we can cover all settings of all triads from 11 two-level factors. Many of these designs can be obtained by a construction due to Roux (1987). If A is a 3-covering design of type $(a, 2^k)$ and B is a 2-covering

Table 5: Sizes of the Best Known 3-Covering $(n, 2^k)$ Designs

k	3	4	5	11	14	16	20	22	28	30	32	40	44	56	64	70	80	121
n	8	8	10	12	16	17	18	19	23	24	25	26	27	31	32	34	35	36

design of type $(b, 2^k)$, and if B' is obtained from B by interchanging the levels of each factor, then the design

$$\begin{matrix} A & A \\ B & B' \end{matrix}$$

is 3-covering of type $(a+b, 2^{2k})$. Roux shows that asymptotically a 3-covering design exists with

$$n = 7.56444 \log_2 k(1 + o(1))$$

Sloane (1993) gives references to results on the cases $q \geq 2$ and $t \geq 4$. etc.. In the general balanced case, i.e. for general $t \geq 2$ with $q_i = q \geq 2$, the problem has been studied extensively, (Sloane gives 51 references), but in very few cases other than the Renyi case are optimal designs known. Very little is known about cases with mixed q 's, or with $t \geq 4$. A survey of the field (with 62 references) is given by Korner and Lucertini (1994). When $q_i = n$, and $k = n + 1$, and when n is a prime number or a power of prime, the standard completely orthogonal designs can also be used for testing. These designs can be generalized by using projective plane constructions. But, typically, they are not directly useful because of severe restrictions on n .

6 Coverage Properties of Some Designs

6.1 Orthogonal arrays

Taguchi (1986) and many other writers have promoted the use of orthogonal arrays, which are balanced fractions of complete factorial designs. In classical terms they are main-effect plans with proportional balance, and hence are $t = 2$ designs. For each such design we can count how many of the possible three-way and four-way combinations are covered. We find that some of these designs are $t = 3$ designs, i.e. they cover all three-factor combinations. However in some cases there are designs that

have better coverage properties than these orthogonal designs. For example, there is an 18-run orthogonal array which is a 2-covering $(18, 2.3^7)$ design. However Table 4 above shows that a 2-covering $(18, 3^{16})$ design exists, and it happens that this can be augmented to form a 2-covering $(18, 2.3^{16})$ design (which we call C_{18}); in fact it can be augmented much further (see below). The $(18, 2.3^7)$ orthogonal array L_{18} (as given by Taguchi) is not a subdesign of C_{18} . We see from Table 6 below that L_{18} covers 70.1% of all possible triads (of its eight factors), while C_{18} covers 57.8% of the possible triads of its 17 factors.

The orthogonal array L_{16} is a 2-covering $(16, 2^{15})$ design, which does not cover all three-way possibilities; Table 5 shows that there is a design with $n = 16$ that covers all triads of 14 two-level factors. Table 6 below gives the t -coverage and t -diversity measures of some orthogonal arrays.

Wang and Wu (1991) give many mixed-radix orthogonal arrays, all of which we here label W . Among these are 2-covering designs of types

- $(24, 4.2^{20})$ (Dey and Ramakrishna 1977),
- $(24, 3.4.2^{13})$,
- $(24, 6.4.2^{11})$ (Agrawal and Dey 1982),
- $(40, 4.2^{36})$ (Dey and Ramakrishna 1977), and
- $(40, 5.4.2^{25})$.

6.2 A Construction

Here is a simple construction for 2-coverage designs of type $(n, 2^k Q)$ when a 2-covering design D of type (n, Q) is known. We simply add to D all the columns of R_n that are 2-covering with respect to the columns of D . Since these columns are columns of R_n , they are necessarily 2-covering amongst themselves. Starting from balanced one-factor designs, this construction gives designs $R(2a, a.2^k)$ with $k = 2^{a-1}$, designs $R(3a, a.2^k)$ with $k = (1/2).3^a C(a, a/2)$ (if a is even), and designs $R(4a, a.2^k)$ with $k =$ the coefficient of x^a in $(4 + 6x + 4x^2)^a$. Starting from balanced designs of type (ab, ab) ,

Table 6: Coverage properties of some systematic and random designs

Name	Type	$t = 2$			$t = 3$				$t = 4$			
		cov	div	c_{rand}	cov	div	c_{rand}	n_{rand}	cov	div	c_{rand}	n_{rand}
L ₄	(4, 2 ³)	1.00	1.00	.684	.500	1.00	.414	6				
R ₆	(6, 2 ¹⁰)	1.00	.667	.822	.688	.917	.551	9	.371	.988	.321	8
C ₆	(6, 3.2 ⁴)	1.00	.800	.744	.577	1.00	.451	9	.268	1.00	.239	7
L ₈	(8, 2 ⁷)	1.00	.500	.900	.900	.900	.656	18	.500	1.00	.403	11
L' ₈	(8, 2 ⁴)	1.00	.500	.900	1.00	1.00	.656		.500	1.00	.403	11
R ₈	(8, 2 ³⁵)	1.00	.500	.900	.767	.767	.656	11	.455	.909	.403	10
C ₈	(8, 4.2 ⁸)	1.00	.611	.811	.643	.857	.530	12	.338	.976	.293	10
L ₉	(9, 3 ⁴)	1.00	1.00	.654	.333	1.00	.288	11	.111	1.00	.106	10
R ₉	(9, 3.2 ³⁶)	1.00	.456	.916	.771	.713	.681	12	.464	.870	.422	11
C ₉	(9, 3 ² 2 ²⁰)	1.00	.514	.894	.731	.741	.641	12	.420	.887	.382	11
R ₁₀	(10, 2 ¹²⁶)	1.00	.400	.944	.831	.664	.737	14	.526	.842	.476	12
R ₁₂	(12, 2 ⁴⁶²)	1.00	.333	.968	.880	.587	.799	16	.590	.787	.539	14
L ₁₂	(12, 2 ¹¹)	1.00	.333	.968	1.00	.667	.799		.688	.917	.539	19
L' ₁₂	(12, 3.2 ⁴)	1.00	.400	.928	.923	.800	.694	27	.527	.983	.420	17
C ₁₂	(12, 6.2 ³²)	1.00	.374	.916	.874	.609	.707	23	.483	.800	.447	14
A ₁₅	(15, 3 ¹²)	1.00	.600	.829	.512	.922	.432	20	.184	.992	.170	17
H ₁₆	(16, 2 ¹⁴)	1.00	.250	.990	1.00	.500	.882		.808	.808	.644	26
L ₁₆	(16, 2 ¹⁵)	1.00	.250	.990	.992	.496	.882	25	.808	.808	.644	26
L ₁₈	(18, 2.3 ⁷)	1.00	.458	.895	.701	.920	.536	29	.267	1.00	.235	21
C ₁₈	(18, 2.3 ¹⁶)	1.00	.480	.887	.578	.816	.512	22	.223	.924	.215	19
L ₂₀	(20, 2 ¹⁹)	1.00	.200	.997	1.00	.400	.931		.882	.706	.725	34
L' ₂₀	(20, 5.2 ⁸)	1.00	.267	.947	.921	.552	.770	39	.622	.829	.507	29
W ₂₄	(24, 4.2 ²⁰)	1.00	.183	.992	.993	.378	.916	58	.841	.668	.706	38
W' ₂₄	(24, 4.3.2 ¹³)	1.00	.201	.986	.961	.421	.877	41	.753	.717	.634	35
W'' ₂₄	(24, 6.4.2 ¹¹)	1.00	.248	.938	.874	.515	.744	42	.578	.800	.475	34
L ₂₇	(27, 3 ¹³)	1.00	.333	.958	.879	.879	.639	56	.329	.988	.285	33
W ₄₀	(40, 4.2 ³⁶)	1.00	.105	.999	.993	.215	.985	50	.934	.414	.884	52
W' ₄₀	(40, 5.4.2 ²⁵)	1.00	.119	.996	.978	.252	.947	58	.871	.487	.786	58

with $a = b = 3$ we find $R(9, 3^2 2^{20})$, with $a = 3, b = 4$ we find $R(12, 3.4.2^{243})$, and with $a = b = 4$ we find $R(16, 4^2.2^{3453})$. Starting from the $(18, 3^{16})$ design mentioned above, we find that no fewer than 16102 columns of R_{18} can be added (66.2% of the 24310 columns of R_{18}).

6.3 Random and Balanced Random Designs

In a random design, in each run, for each factor we simply assign the level at random (independently for each run). Clearly, deleting any repeated rows will give greater efficiency, but this will not help much since such repetitions are very unlikely. It is helpful to assign the levels at random subject to the constraint that in each column each level occurs as nearly as possible the same number of times. We recall that Satterthwaite (1959) proposed to use random constructions for classical purposes, where they are not very effective. However we will see that they perform well as coverage designs as long as complete t -coverage is not required. Lin (1995) has used the balanced random idea to construct supersaturated designs.

6.4 Some Numerical Comparisons

Table 6 summarizes the coverage properties (up to $t = 4$) of some of the designs we have mentioned. All these designs have 100% 2-way coverage. Designs $L_4, L_8, L_9, L_{12}, L'_{12}, L_{16}, L_{18}, L_{20}$, and L'_{20} , are all orthogonal arrays, of various shapes. L_8, L_{12}, L_{16} , and L_{20} are also Hadamard designs, obtained by deleting one column from a Hadamard matrix. There are five types of 16-row Hadamard matrices; their 3-coverages range from .962 (for the ‘‘Sylvester’’ type, the fourfold Kronecker product of H_2), to .992 (for types $C(B_3)$ and $C(B_4)$ in the notation of Assmus and Key (1992)). Remarkably, all five types have the same 4-coverage. The three types of 20-row Hadamard design all have the same coverage properties (up to $t = 4$). L'_8 is a subdesign of L_8 ; it is included because it has perfect 3-factor coverage. Designs R_6, R_8, R_9, R_{10} , and R_{12} are ‘‘Renyi’’ designs, known to be optimal 2-covering designs. Designs C_6, C_8, C_9 , and C_{12} are obtained using the construction described above, starting from balanced designs of shapes $(6,3)$, $(8,4)$, $(9,3^2)$ and $(12,6)$ respectively. Designs $W_{24}, W'_{24}, W''_{24}, W_{40}$ and W'_{40} are taken from Wang and Wu (1991). The designs

A_{15} and A_{18} are new 2-covering designs for 3-level factors; they use L_9 (a 3-covering design) as a building block, and are given in the Appendix. Finally, H_{16} is also new, being obtained by deleting the first and ninth columns from a certain Hadamard matrix ($C(B_2)$ in the notation of Assmus and Key (1992)).

Table 6 also gives some properties of some random designs that could be used as alternatives to these systematic designs. For $t = 2, 3, 4$ we show c_{rand} , the expected t -coverage of the random design of the same size as the systematic design. Also, if the systematic design has coverage strictly less than unity, we show n_{rand} , the size of the smallest random design for which the expected t -coverage exceeds that of the systematic design.

From this Table we see that orthogonal arrays are typically very inefficient as 2-covers, but are very effective as 3-covers; in all cases random designs are considerably less efficient as 3-covers, but are often nearly as good as the systematic designs as 4-covers.

Similar calculations show that balanced random designs are slightly more efficient than completely random ones, but are still less efficient than the systematic designs.

7 Constructing Factor-Covering Designs

We have seen above that factor-covering designs with moderate efficiency can be constructed at random. Sherwood (1994) has written (in C) a program called CATS (Constrained Array Test System) which finds covering designs using a greedy search algorithm. Given Q , a list of all feasible runs is generated. (there are $|Q|$ of them). If some runs are prohibited, they are excluded from this list. Also, given t , a list of all possible t -factor settings is generated. This is F_t in the notation of Section 3. We want a design that covers each of the settings in F_t . At each stage, after some number of runs has been chosen, so that some fraction of these settings have been covered, each of the unused runs is examined, and the number of additional t -factor settings that are covered is computed. The algorithm selects the run that maximizes the number of t -way settings that are covered. (If more than one run achieves the maximum, the first one encountered is

chosen.) This procedure continues until all the t -factor settings have been covered.

Since the possible runs are tested in sequence, the user can start with a list of runs that takes the importance of the factors into account, so that the more critical settings will be covered early. To handle large problems where exhaustive search of all the possibilities is not feasible, the algorithm starts by choosing a subset of the factors, say f_1 of them, and a covering design D_1 for these factors is constructed. Then an additional subset of f_2 factors is considered, and a list of feasible runs is generated by concatenating the runs of D_1 with all possible settings of these new factors. A covering design D_2 for the $f_1 + f_2$ factors is generated by selecting from this list. This construction is iterated until all the factors have been entered. This strategy ensures that the size of the list of runs that need to be considered remains manageable. The present version of CATS does not use randomization. It is not clear how much might be gained (in terms of number of runs) by using randomization throughout the algorithm.

Dalal and Patton (1993) proposed the Automatic Efficient Test Generator (AETGTM) system¹ which was substantially extended with several new functionalities by Cohen, Dalal, Kajla and Patton (1994) and Cohen, Dalal, Fredman and Patton (1997). It has similar functionality as CATS, but, it uses a variety of algorithms. One of the algorithms in the original 1992 prototype by Dalal and Patton is locally greedy: i) to start a new test case, it selects an uncovered t -combination of factors for a t -combination of levels which has the maximum number of t -combinations of levels not covered across all factors, ii) having selected the first t -combination, it selects another factor at random, and a level for this factor so as to minimize the number of remaining uncovered t -combinations. This process is iterated until all t -combinations are covered. Several variations of this algorithm are possible. Some of the theoretical properties of this algorithm are given by Cohen, Dalal, Fredman and Patton (1997).

The AETG system seems to be more more efficient than CATS. For example, for 20 factors, each with 10 levels, CATS found a $(240, 10^{20})$ design; the AETG system found a $(181, 10^{20})$ design (Cohen and Fredman, 1996).

¹AETG is a trademark of Bellcore. Covered under Bellcore's US patent #5,542,043

8 An Application of the AETG System

Here we describe one simple application of AETG to a problem described by Williams and Roberts (1996) on network interface testing. Specifically, they describe a situation that involves 5 factors: an originating party calls a receiving party from a phone, through an originating interface, through a switch and a receiving interface to a receiving phone. The specific levels for these 5 parameters are Originating and Receiving Phone Types (RES, BUS, COIN, ISDN), Interface between a phone and a switch (A and B), and switch market (CANADA, US, MEXICO, FRANCE, UK). RES and BUS stand for Residential and Business phones. There are constraints requiring that i) the ISDN line can only use Interface B, and ii) RES and BUS can only use Interface A on the near end, though they can use either A or B on the far end. For this model they collapsed the parameters to a set of valid configurations in the cross product of Phone Types and Interface on the originating and the receiving side. This gave 3 parameters with 5 levels each, and they proposed 5x5 Latin Square design for testing this, i.e. 25 test cases. Using the AETG system and taking the constraints into account directly we found the set of 20 test cases given in Table 7.

These test cases obey all the constraints and cover all the valid pairs of levels. There have been many other applications of AETG system- most are more complex- some involving as many as 120 parameters. Amongst publicly reported applications by people other than developers of AETG system are Burroughs, Jain and Erickson (1994) and Burr and Young (1997). The first one reports three applications of the AETG system involving protocol performance testing. These involve designs of types $(42, 7.6.3.2^2)$, $(18, 6.2.3^2)$, and $(12, 4.3.2)$ respectively. Burr and Young (1997) report another interesting application at Nortel.

9 Discussion on Effectiveness

Clearly we need to know what values of t are most important in practice. According to Nair, James, Ehrlich, and Zavallos (1997), in the (small) cases they looked at, designs that cover all 2-factor settings are effective in detecting most faults; this suggests that in these cases at least,

Table 7: Test cases for testing switches made for different countries

Orig. Phone	Inter-face	Switch Market	Inter-face	Rec. Phone
BUS.	A	MEXICO	A	BUS.
COIN	B	UK	B	COIN
ISDN	B	USA	A	RES.
RES.	A	USA	B	ISDN
COIN	A	FRANCE	A	RES.
RES.	A	CANADA	A	COIN
ISDN	B	CANADA	B	ISDN
ISDN	B	MEXICO	B	COIN
BUS.	A	FRANCE	B	ISDN
RES.	A	UK	A	RES.
ISDN	B	FRANCE	A	BUS.
COIN	A	MEXICO	B	ISDN
BUS.	A	CANADA	A	RES.
COIN	B	USA	A	BUS.
ISDN	B	UK	B	ISDN
BUS.	A	USA	B	COIN
RES.	A	CANADA	A	BUS.
RES.	A	FRANCE	A	COIN
RES.	A	MEXICO	A	RES.
BUS.	A	UK	A	BUS.
COIN	A	CANADA	B	ISDN

the model (Section 3) in which the only non-zero p is p_2 might be appropriate. We could estimate p_2 from these experiences and hence estimate how many faults to expect in similar systems. Also, if this model does seem to represent reality, we can choose to use $t = 2$ designs with confidence that most faults will be found. Further evidence for $t = 2$ comes from several papers on AETG system (Cohen et al, 1994, 1996, 1997). These authors report that in testing ten screens which had already gone through a first level of system test, the pairwise strategy found on average 10 problems per screen. Further, in testing an intelligent network service, test cases designed by AETG found 43 problems. Further, in testing a Unix command like “sort”, the pairwise design produced 96% software coverage (the percent of software statements touched by the test cases). For more complex systems, in many instances, during system testing the AETG system produced

software coverage of around 80-90%. This can be compared with the study of Plwowarski, Obha, and Caruse (1993) which suggests that during system testing software coverage beyond 50-60% is difficult to achieve.

On the other hand, Dunietz, Ehrlich, Szablak, Mallows, and Iannino (1997) found that for a particular software product, $t = 3$ designs sufficed to achieve effective software coverage (i.e. this ensured that all blocks of code were exercised), though higher values of t were required to cover all paths through the code. Dunietz, Ehrlich, and Mallows (1997) analyzed several features of a system for access to a repository of reusable software assets. They found seven faults. One was trivial, and would have been found by a $t = 1$ design; an estimate of the p_1 parameter in the model of Section 3 is .248. Two other faults needed a $t = 2$ design; for these, estimates of p_2 are .081 and .0045 respectively. Finally, four of the faults would have been found by a $t = 3$ design, and estimates of p_3 are .00073, .0034, .0014, and .00051 respectively.

10 Open Issues

The main outstanding technical problem is that of constructing efficient covering designs in an unified manner. It is not clear that combinatorial complexity theory has anything to say about how difficult this problem is. How much does randomization help, if at all? If we construct many random designs and choose the best, how close will we get to optimality? The asymptotic result (1) needs to be extended to $t \geq 3$, and to mixed-radix cases.

In this paper we have assumed that software works in a fixed environment, so that replication is wasteful. If the software works in a random environment (e.g. another program is running concurrently which affects the environment), what can be said about probability of detection of a fault? Further, we have concentrated on achieving 100% t -coverage; but how about choosing two fractions f and p , and asking for $100f\%$ coverage with probability p ? These issues have been discussed by Mallows (1997), but extending the asymptotic results in this paper to these issues would be worthwhile.

Another practical problem is that often certain constraints are imposed on the input space. For example, suppose there are three possible inputs for a transaction-cash payment (yes, or blank), credit card payment (yes, or

blank), and credit card number (16 bytes alpha numeric). Then the natural constraint is that if the cash input is yes, the credit card inputs should be blank, and vice versa. While the CATS and AETG systems are able to deal with these problems, it would be useful to introduce such constraints into the algebraic structure.

Acknowledgements

We thank D. Cohen, W. K. Ehrlich, M. Fredman, J. Musa, G. Patton, G. B. Sherwood and P. Seymour for helpful discussions. AETG is a trademark of Bellcore, covered under Bellcore's US patent 5,542,043.

References

- Agrawal, V. and Dey, A. (1982), "A note on orthogonal main effect plans for asymmetrical factorials," *Sankhyā*, Ser. B, 44, 278–282.
- Assmus, E. F. and Key, J. D. (1992), "Hadamard matrices and their designs; A coding theoretic approach," *Trans Amer. Math Soc.*, 330, 269–293.
- Beizer, B. (1995), *Black-Box Testing*, New York: Wiley.
- Booth, E. H. and Cox, D. R. (1962), "Some systematic supersaturated experimental designs," *Technometrics*, 4, 489–495.
- Box, G. E. P. and Tyssedal, J. (1996), "Projective properties of certain orthogonal arrays," *Biometrika*, 83, 950–955.
- Brownlie, R., Prowse, J., and Phadke, M. S. (1992), "Robust testing of AT&T PMX/StarMAIL using OATS," *AT&T Technical Journal*, 71(3), 41–47.
- Burr, K. and Young, W. (1997), "Test Acceleration (Automatic Efficient Testcase Generation)," Presented at Nortel Design Forum, Dec 3–5.
- Burroughs, K., Jain, A., and Erickson, R. L. (1994), "Improved quality of protocol testing Through techniques of experimental design." *Supercomm/ICC '94 (IEEE International Conf. on Communications)*, 745–752.
- Cohen, D. M., Dalal, S. R., Kajla, A., and Patton, G. C. (1994), "The automatic efficient test Generator (AETG) system," *Proceedings 5th Intl. Symp. Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos, CA.
- Cohen, D. M., Dalal, S. R., Parelius, J., and Patton, G. C. (1996), "The Combinatorial Approach to Automatic Test Generation," *IEEE Software*, 13, 83–89.
- Cohen, D. M., Dalal, S. R., Fredman, M., and Patton, G. C. (1997), "The AETG System: An Approach to Testing Based on Combinatorial Design," *IEEE Transactions of Software Engineering*, 23
- Cohen, D. M. and Fredman M (1996), *New Techniques for Designing Qualitatively Independent Systems*, Rutgers University Technical Report, DCS-96-114
- Dalal S. R., Horgan J. R., Kettnering J. R. (1994), "Reliable software and communication II, Controlling the software development process," *IEEE J. Selected Areas in Communications*, 12, 33–39.
- Dalal, S. R. and Mallows, C. L. (1988), "When should one stop testing software?" *J. American Statist. Assoc.*, 83, 872–879.
- Dalal S. R. and McIntosh A. M.(1994), "When to stop testing for large software systems with changing code," *IEEE Trans. of Software Eng.*, 20, 318–323.
- Dalal, S. R. and Patton G. C. (1993), *Automatic Efficient Test Generator (AETG): A test generation system for Screen Testing, Protocol Verification and Feature Interactions Testing*, Bellcore Technical Memo.
- Dey, A. and Ramakrishna, G. V. S. (1977), "A note on orthogonal main effect plans," *Technometrics*, 19, 511–512.
- Dunietz, I. S., Ehrlich, W. K., Hecht, D. A., Olson, C. H., Prasad, R., and Szablak, B. D. (1995), "Deriving efficient test designs through software instrumentation and software visualization," unpublished TM, AT&T Bell Laboratories.
- Dunietz, I. S., Ehrlich, W. K., and Mallows, C. L. (1997), "Experience with covering designs for testing software," to appear, *Proc. Computing Sect. ASA*.
- Dunietz, I. S., Ehrlich, W. K., Szablak, B. D., Mallows, C. L., and Iannino, A. (1997), "Applying systematic t -factor covering experimental designs to software testing," submitted for publication.
- Economist Magazine (1997), "The Millennium Bug", October 4th, 1997 issue, pp. 25–29.
- Gargano, L., Korner, J., and Vaccaro, U., (1993), "Spener capacities," *Graphs and Combinatorics*, 9, 31–46.

- Godbole, A. P., Skipper, D. E., and Sunley, R. A. (1995), “ t -covering arrays: upper bounds and approximations,” Preprint, Michigan Technological University.
- Humphrey, W. S. (1989), *Managing the Software Process*, Addison Wesley, Reading, Mass
- Jorgensen, P. (1995), *Software Testing, A Craftsman’s Approach*, CRC Press, New York.
- Katona, G. O. H (1973), “Two applications (for search theory and truth functions) of Sperner type theorems,” *Periodica Math. Hung.*, 3, 19–26.
- Kleitman, D. J. and Spencer, J. (1973), “Families of k -independent sets,” *Discrete Math.*, 6, 255–262.
- Korner, J. and Lucertini, M. (1994), “Compressing inconsistent data,” *IEEE Trans. Information Theory*, 40, 706–715.
- Lin, D. K. J. (1993), “A new class of supersaturated designs,” *Technometrics*, 35, 28–31.
- Lin, D. K. J. (1995), “Generating systematic supersaturated designs,” *Technometrics*, 37, 213–225.
- Mallows, C. L. (1996), “Covering designs in random environments,” Submitted to *Festschrift for J. W. Tukey*, ed. S. Morgenthaler.
- Miller, E. (1991), Panel Discussion, Testing and Verification Analysis Conference, (TVA), Victoria, British Columbia.
- Mandl, R. (1985), “Orthogonal Latin Squares: An Application of Experimental Design to Compiler Testing,” *Communications of the ACM*, 28(10), 1054–1058.
- Myers, G. J. (1979), *The Art of Software Testing*, John Wiley, New York.
- Mills, W. and Mullin, R. (1992), “A survey on coverings and packings,” pp. 371–299 in *Contemporary Design Theory: a Collection of Surveys*, ed. J. Dinitz and D. Stinson, Wiley, New York.
- Nair, V. N., James, D. A., Ehrlich, W. K., and Zevallos, J. (1998), “A statistical assessment of some software testing strategies and application of experimental design techniques,” to appear, *Statistica Sinica*.
- National Research Council (1996), *Statistical Software Engineering*, NRC report by a panel on Statistical Methods in Software Engineering.
- Plowowski, P, Obha, M. and Caruse, J (1993), “Coverage Measurement Experience During Function Test,” *Proc. 15th International Conference on Software Engineering*, IEEE CS Press, Los Alamitos, Calif., 287–303.
- Renyi, A. (1971), *Foundations of Probability*, Wiley, New York.
- Roux, G. (1987), k -propriétés dans des Tableaux de n Colonnes; Cas Particulier de la k -Surjectivité et Dica/de la k -permutivité, Ph.D. dissertation, University of Paris.
- Satterthwaite, F. (1959), “Random balance experimentation (with discussion),” *Technometrics*, 1, 111–137.
- Sherwood, G. B. (1994), “Effective testing of factor combinations,” *Proceedings of the Third International Conference on Software Testing, Analysis and Review*, Washington D.C. May 8–12, 1994.
- Singpurwalla, N. D. (1991), “Determining an optimal time interval for testing and debugging software,” *IEEE Trans. Software Eng.*, 17(4), 313–319.
- Sloane, N. J. A. (1993), “Covering arrays and intersecting codes,” *J. Combinatorial Designs*, 1, 51–63.
- Taguchi, G. (1986), *Introduction to quality engineering*, Kraus International Publications, White Plains, New York.
- Tatsumi, K., Watanabe, S., Takeuchi, Y., and Shimokawa, H. (1987), “Conceptual support for test case design,” *Proceedings 11-th Computer Software and Applications Conference (Compsac 87)*, 285–290.
- Wang, J. C. and Wu, C. F. J. (1991), “An approach to the construction of asymmetrical orthogonal arrays,” *J. American Statistical Association*, 8, 450–456.
- Williams, A. L., Probert R. L.(1996), “A practical strategy for testing pairwise coverage at network interfaces,” *Proceedings 7th Intl. Symp. Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos, CA.
- Wu, C. F. J. (1993), “Construction of supersaturated designs through partially aliased interactions,” *Biometrika*, 80, 661–669.
- Zeitler, D. (1991), “Statistical methodology for software systems testing,” *Proceedings of the 23rd Interface Symposium*, The Interface Foundation of North America Inc. pp. 110–113.