Constructing Strength Three Covering Arrays with Augmented Annealing

Myra B. Cohen, Charles J. Colbourn, Alan C.H. Ling [a,b,c]

[a] *Dept. of Computer Science*
University of Auckland
Private Bag 92019
Auckland, New Zealand
myra@cs.auckland.ac.nz

[b] *Dept. of Computer Science and Engineering*
Arizona State University
P.O. Box 875406
Tempe, Arizona 85287
charles.colbourn@asu.edu

[c] *Dept. of Computer Science*
University of Vermont
Burlington, Vermont 05045
aling@emba.uvm.edu

Abstract

A *covering array* $CA(N; t, k, v)$ is an $N \times k$ array such that every $N \times t$ sub-array contains all t-tuples from v symbols *at least* once, where t is the *strength* of the array. One application of these objects is to generate software test suites to cover all t-sets of component interactions. Methods for construction of covering arrays for software testing have focused on two main areas. The first is finding new algebraic and combinatorial constructions that produce smaller covering arrays. The second is refining computational search algorithms to find smaller covering arrays more quickly. In this paper, we examine some new cut-and-paste techniques for strength three covering arrays that combine recursive combinatorial constructions with computational search, *augmented annealing*. This method leverages the computational efficiency and optimality of size obtained through combinatorial constructions while benefiting from the generality of a heuristic search. We present a few examples of specific constructions and provide new bounds for some strength three covering arrays.

Key words: software testing, combinatorial design, covering array, heuristic search, simulated annealing, orthogonal array, generalized Hadamard matrix

1 Introduction

Component based software development poses many challenges for the software tester. Interactions among components are often complex and abundant. Components may not be designed with the final product in mind which leaves them prone to unexpected interaction faults. Ideally we want to test all possible interactions, but this is usually infeasible either time-wise or cost-wise. We are, therefore, interested in generating test suites that provide coverage of as many interactions as possible.

Suppose we have 20 components. If two of these have four possible configurations, while the rest have three, we have $4^2 \times 3^{18}$ or 6,198,727,824 possible interactions. We can, however, cover all of the two-way interactions among these components with as few as 19 tests. Likewise, we can cover the three way interactions with only 90 tests. Recently, these methods have been applied to the generation of software test suites allowing one to guarantee certain interaction coverage in software systems [4, 5, 6, 7, 12, 13, 20, 21, 22, 23].

Table 1 shows a small example of four components with three configurations each. In this scenario we have $3^4 = 81$ possible interactions. We are testing software components for a new integrated RAID controller. If it is not possible to test all 81 interactions we can instead decide to test all pairs or triples of interactions. For instance the first test case, (RAID 5, Novell, 128 MB, Ultra 160-SATA), covers six pairs of interactions (RAID 5 with Novell, RAID 5 with 128 MB of memory, RAID 5 with an Ultra 160-SATA disk interface, Novell with 128 MB of memory, Novell with an Ultra 160-SATA interface, and 128 MB of memory with an Ultra 160-SATA interface) or four triples of interactions (RAID 5 and Novell with 128 MB, RAID 5 and Novell with Ultra 160-SATA, RAID 5 and 128 MB with Ultra 160-SATA, and Novell and 128 MB with Ultra 160-SATA). All two way interactions can be covered with nine test cases or all three way interactions with the 27 test cases shown in Table 2.

At the current time there are two distinct areas of active research on combinatorial designs for software testing. The mathematics community is focusing on building smaller designs of higher interaction strength [2, 3, 16, 17, 18]. The software testing community is focusing on greedy search algorithms to build these in a more flexible environment, one that more closely matches real testing needs [4, 5, 12, 13, 21, 20, 23]; in addition, more powerful search techniques such as simulated annealing have been employed recently [6, 7]. Ideally we would like to combine these ideas and build higher strength interaction tests that are minimal and efficient to generate. As the methods of building covering arrays for testing are varied, a trade off must occur between computational power and the cost of running the final test suites. In this paper we

Component			
RAID Level	**Operating System**	**Memory Config**	**Disk Interface**
RAID 0	Windows XP	64 MB	Ultra-320 SCSI
RAID 1	Linux	128 MB	Ultra-160 SCSI
RAID 5	Novell Netware 6.x	256 MB	Ultra-160 SATA

Table 1
RAID integrated controller system: 4 components, each with 3 configurations

examine some methods of combining both computational search and recursive combinatorial construction to efficiently build optimal test suites.

Component								
RAID Level	**Operating System**	**Memory Config**	**Disk Interface**		**RAID Level**	**Operating System**	**Memory Config**	**Disk Interface**
RAID 5	Novell	128 MB	Ultra 160-SATA		RAID 1	Linux	64 MB	Ultra 320
RAID 5	Novell	64 MB	Ultra 320		RAID 5	Novell	256 MB	Ultra 160-SCSI
RAID 1	Novell	256 MB	Ultra 320		RAID 1	Linux	256 MB	Ultra 160-SCSI
RAID 1	XP	128 MB	Ultra 320		RAID 5	XP	256 MB	Ultra 320
RAID 5	Linux	256 MB	Ultra 160-SATA		RAID 5	XP	64 MB	Ultra 160-SATA
RAID 1	Novell	128 MB	Ultra 160-SCSI		RAID 0	Novell	256 MB	Ultra 160-SATA
RAID 0	Linux	64 MB	Ultra 160-SATA		RAID 0	XP	256 MB	Ultra 160-SCSI
RAID 0	XP	128 MB	Ultra 160-SATA		RAID 0	Linux	128 MB	Ultra 160-SCSI
RAID 1	Linux	128 MB	Ultra 160-SATA		RAID 1	XP	64 MB	Ultra 160-SCSI
RAID 0	Novell	128 MB	Ultra 320		RAID 5	XP	128 MB	Ultra 160-SCSI
RAID 5	Linux	64 MB	Ultra 160-SCSI		RAID 0	XP	64 MB	Ultra 320
RAID 5	Linux	128 MB	Ultra 320		RAID 1	Novell	64 MB	Ultra 160-SATA
RAID 0	Novell	64 MB	Ultra 160-SCSI		RAID 0	Linux	256 MB	Ultra 320
RAID 1	XP	256 MB	Ultra 160-SATA					

Table 2
Test suite covering all 3-way interactions for Table 1

2 Covering Arrays and Heuristic Search

The problems faced in software interaction testing are not unique. Similar problems exist for testing in other disciplines such as agriculture, pharmaceuticals, manufacturing and medicine [14]. The primary combinatorial objects used to satisfy the coverage criteria for these types of problems are orthogonal arrays and covering arrays. We begin with a few definitions and then describe how these objects can be applied to software testing.

An *orthogonal array* $OA_\lambda(N; t, k, v)$ is an $N \times k$ array on v symbols such that every $N \times t$ sub-array contains all ordered subsets of size t from v symbols *exactly* λ times. When λ is one we drop the subscript. Orthogonal arrays have the property that $\lambda = \frac{N}{v^t}$. We do not need such a stringent object for software testing. In fact orthogonal arrays are too restrictive since they only exist for

certain values of t, k, v. Instead we can use a covering array that allows some duplication of coverage.

A *covering array* $CA_\lambda(N; t, k, v)$ is an $N \times k$ array such that every $N \times t$ sub-array contains all ordered subsets from v symbols of size t *at least* λ times. When λ is one we omit the subscript. The *covering array number* $CAN(t, k, v)$ is the minimum number N of rows required to produce a $CA(N; t, k, v)$. For example, $CAN(2, 5, 3) = 11$ [3]. In a covering array, $CA(N; t, k, v)$, t is the *strength*, k the *degree* and v the *order*.

We map a covering array to a software test suite as follows. In a software test we have k *components* or *fields*. Each of these has v *configurations* or *levels*. A test suite is an $N \times k$ array where each row is a test case. Each column represents a component and the value in the column is the particular configuration. In Table 2 a $CA(27; 3, 4, 3)$ is given. Each component is represented by one column and each row is an individual test case of the test suite.

In software systems, the numbers of configurations for each component vary in size. We define a more general object to describe this variability (see [7] for a more in-depth discussion). A *mixed level* covering array, $MCA(N; t, k, (v_1, v_2, ..., v_k))$, is an $N \times k$ array on v symbols, where $v = \sum_{i=1}^{k} v_i$, with the following properties:

(1) Each column i $(1 \leq i \leq k)$ contains only elements from a set S_i with $|S_i| = v_i$.
(2) The rows of each $N \times t$ sub-array cover all $t-$tuples of values from the t columns at least once.

When $t = 3$, the combinatorial research illustrates both of the depth of the connection with combinatorial configurations and the difficulties that these pose for software testers. The techniques applied to date when $t = 3$, at least for *small* covering arrays, range from very simple construction methods such as identifying distinct symbols to form a single symbol, through to more complex *cut-and-paste* constructions using smaller covering arrays, and ultimately sophisticated recursive constructions that combine small covering arrays but also employ related combinatorial objects such as perfect hash families [2]. While the more sophisticated constructions yield substantially smaller covering arrays when they can be applied, these same constructions do not apply as generally as we require. For a summary of known results when $t = 3$ see [3].

Computational search techniques to find covering arrays include greedy algorithms and standard combinatorial search techniques such as simulated annealing [4, 7, 20, 23]. We use simulated annealing, a search technique for solving combinatorial optimization problems, that yields good general results for finding minimal test suites especially when the problem size is relatively

small [7].

In simulated annealing a search problem can be specified as a set Σ of feasible solutions (or states) together with a cost $c(S)$ associated with each feasible solution S. An optimal solution corresponds to a feasible solution with overall (i.e. global) minimum cost. We define a feasible solution $S \in \Sigma$, a set T_S of transformations (or transitions), each of which can be used to change S into another feasible solution S'. The set of solutions that can be reached from S by applying a transformation from T_S is the *neighbourhood* $N(S)$ of S.

To start, we randomly choose an initial feasible solution. At each *trial*, we select a transition to a neighbour at random. If the neighbour has lower or equal cost, we accept the transition. If the transition results in a feasible solution S' of higher cost, then S' is accepted with probability $e^{-(c(S')-c(S))/T}$, where T is the controlling temperature of the simulation. The temperature is lowered in small steps to allow the system to approach "equilibrium" at each temperature through a sequence of trials at this temperature. Usually this is done by setting $T := \alpha T$, where α (the *control decrement*) is a real number slightly less than one. After an appropriate stopping condition is met, the current feasible solution is taken as the solution of the problem at hand. Allowing a move to a worse solution helps to keep the solution from being stuck in a bad configuration, while continuing to make progress. The algorithm stops once a feasible solution of cost zero is obtained or the current solution is frozen. See [6, 7] for a more detailed description of using simulated annealing to find covering arrays.

Simulated annealing performs well when the search space is small and there are abundant solutions. As the search space increases and the density of potential solutions becomes sparser the algorithm may fail to find a good solution or may require extremely long run times. Careful tuning of the parameters of temperature and cooling improves upon the results, but at a potential computational cost. Cohen *et al.* present results suggesting that annealing works well for covering arrays, often produces smaller test suites than other computational methods, and sometimes improves upon combinatorial constructions. It fails to match the known constructions for larger problems, especially when $t = 3$ [7].

Recursive and direct combinatorial constructions often provide a better bound in less computational time than heuristic search. However, they are not as general and must be tailored to the problem at hand. An in-depth knowledge is often needed to decide which construction best suits a particular problem. We develop a new strategy to take advantage of the strengths of each, which we call *augmented annealing*. The idea of using small building blocks to construct a larger array is used often in combinatorial constructions. We refer to these techniques in general as *cut-and-paste* methods. Often techniques to obtain a

general solution result in objects that are larger than need be. If our only aim is to construct an individual object we can relax the construction and build an object that fits our needed criteria. In the next sections we use combinatorial constructions and augment them with heuristic search to allow one to construct an array. We have used this method successfully to construct objects with lower bounds than that of simulated annealing alone and in many cases have improved upon results for known combinatorial constructions [8].

We outline the primary ideas in augmented annealing next. Consider a typical recursive construction. The problem is decomposed by using a "master" structure that is used to determine the placement of certain "ingredients". In this prototype scheme, a number of fatal problems can arise. A decomposition imposed by the master may not cleanly separate the ingredients, so that ingredients overlap or interact. The character and extent of the interaction results in either a specialized definition of allowed ingredients, or (as in our covering problem) additional coverage not required in the problem statement. Combinatorial constructions focus on proving general results, and hence often permit an overlap that is asymptotically small. However, for instances that are themselves small, the overlap can mean the difference between a good solution and a poor one.

Even more severe problems arise. It may happen that in a combinatorial construction, we have no general technique for producing the needed ingredients. When this occurs, the construction simply fails, despite its "success" at constructing a large portion of the object sought. When this happens, current techniques abandon the combinatorial construction and employ computational search.

Augmented annealing suggests a middle road. We use a combinatorial construction to decompose the problem, but then use simulated annealing (or any other search technique) to:

(1) produce ingredients for which no combinatorial construction is known;
(2) minimize overlap between and among ingredients; and
(3) complete partial structures (*seeded tests*) when no combinatorial technique for completion is available.

This enables us to use combinatorial decompositions to reduce a problem to a number of smaller subproblems, on which simulated annealing can be expected to be both faster and more accurate than on the problem as a whole. By having simulated annealing use knowledge about which t-tuples really need to be covered, we avoid much duplicate coverage in general constructions.

In the remainder of the paper, we illustrate this idea using three combinatorial constructions; the first class of constructions is discussed in [8], while the second is a new general construction.

3 Cut-and-Paste-Constructions

We present several combinatorial constructions in this section that involve decomposing the covering array into smaller objects. We extend the "traditional" approach in recursive constructions by allowing small pieces to be built using computational search.

3.1 Ordered Design Construction

An *ordered design* $OD_\lambda(t,k,v)$ is a $k \times \lambda \cdot \binom{v}{t} \cdot t!$ array with v entries such that

(1) each column has v distinct entries, and
(2) every t columns contain each row tuple of t distinct entries precisely λ times.

When $\lambda = 1$ we write $OD(t,k,v)$. An $OD(3, q+1, q+1)$ exists when q is a prime power [10]. We use an ordered design as an ingredient for building a $CA(3, q+1, q+1)$ since it already covers all triples with distinct entries, having the minimal number of blocks. This handles many but not all of the triples required. The covering array is completed by covering the remaining triples. We describe a general construction next.

Construction 1 $CA(3, q+1, q+1) \leq q^3 - q + \binom{q+1}{2} \times CA(3, q+1, 2)$ *when* q *is a prime power.*

To create a $CA(3, q+1, q+1)$ begin with a $OD(3, q+1, q+1)$ of size $N_3 = (q+1) \times q \times (q-1)$. This covers all triples of the form (a,b,c) where $a \neq b \neq c \neq a$. To complete the covering array we need to cover all of the triples of the form (a,a,b), (a,b,b), (a,b,a) and (a,a,a). These are exactly the triples covered by a $CA(N_2; 3, q+1, 2)$ on symbol set $\{a,b\}$. Since a and b can be any of $\binom{q+1}{2}$ combinations we append $\binom{q+1}{2}$ $CA(N_2; 3, q+1, 2)$s to the N_3 rows of the ordered design. This gives us a $CA(3, q+1, q+1)$.

Unnecessary coverage of triples occurs. In fact, any triple of the form (a,a,a) is covered at least q times rather than once. We therefore relabel entries in the $CA(N_2; 3, q+1, 2)$s to form a constant row; deleting these reduces the number of rows required by $\binom{q+1}{2}$. We can save even more:

Construction 2 $CA(3, q+1, q+1) \leq q^3 - q + \binom{q+1}{2} \times CA(3, q+1, 2) - (q^2 + 2q + 1)$ *when* q *is a prime power and there are two disjoint rows in the* $CA(3, q+1, 2)$.

In Construction 1 we exploit overlap in coverage of triples that occurs if each of the $CA(N_2; 3, q+1, 2)$s has two disjoint rows. In this case we remap the two disjoint rows, without loss of generality, to the form $(a, a, ..., a)$ and $(b, b, ..., b)$. We remove the $2 \times \binom{q+1}{2} = q^2 + q$ rows and add back in $q+1$ rows of the form $(a, a, ..., a)$.

We give an example using $CA(3, 6, 6)$. The ordered design has 120 rows. There are 15 combinations of two symbols. In Construction 1, we create a $CA(3, 6, 2)$ with 12 rows. We therefore add back in 180 rows. This gives us a $CA(3, 6, 6)$ of size 300. This is smaller than the bound reported by a construction in [3], and matches that found by annealing in [7]. Removing 15 constant rows lowers this bound to 285. For Construction 2, we find a $CA(12; 3, 6, 2)$ having two disjoint rows (see Table 3). Therefore we remove 30 rows of the type $(a, a, ..., a)$ for a total of 270 rows. We add back in six rows, one for each symbol, to achieve a covering array of size 276. This improves on both reported bounds above.

Let us generalize further. A (2,1)-*covering array*, denoted by $TOCA(N; 3, k, v; \sigma)$ is an $N \times k$ array containing σ or more constant rows, in which every $N \times 3$ subarray contains every 3-tuple of the form (a, a, b), (a, b, a), and (b, a, a) with $a \neq b$, and contains every triple of the form (a, a, a). $TOCAN(3, k, v; \sigma)$ denotes the minimum number N of rows in such an array.

A set $\mathcal{B}$ of subsets of $\{1, \ldots, k\}$ is a *linear space* of order k if every 2-subset $\{i, j\} \subseteq \{1, \ldots, k\}$ appears in exactly one $B \in \mathcal{B}$.

Construction 3 *Let q be a prime power. Let $\mathcal{B} = \{B_1, \ldots, B_b\}$ be a linear space on $K = \{1, \ldots, k\}$. Let $\emptyset \subseteq L \subseteq K$. Suppose that for each $B_i \in \mathcal{B}$ there exists a $TOCA(N_i; 3, q+1, |B_i|; |B_i \cap L|)$. Then there exists a $CA(q^3 - q + |L| + \sum_{i=1}^{b}(N_i - |B_i \cap L|); 3, q+1, q+1)$.*

We start with an $OD(3, q+1, q+1)$ and for each $B_i \in B$, we construct the TOCA on the symbols of B_i with the constant rows (to be removed) on the symbols of $B_i \cap L$. Then $|L|$ constant rows complete the covering array.

3.2 Roux-type Constructions

In [16], a theorem from Roux's Ph.D. dissertation is presented.

Theorem 1 $CAN(3, 2k, 2) \leq CAN(3, k, 2) + CAN(2, k, 2)$.

Proof. To construct a $CA(3, 2k, 2)$, we begin by appending two $CA(N_3, 3, k, 2)$s side by side. We now have a $N_3 \times 2k$ array. If one chooses any three columns whose indices are distinct modulo k, then all triples are covered. The remaining selection consists of a column x from among the first k, its copy among

the second k, and a further column y. When the two columns whose indices agree modulo k are to share the same entry, such a triple is also covered. The remaining triples are handled by appending two $CA(N_2, 2, k, 2)$s side by side, the second being the bit complement of the first. Therefore if we choose two distinct columns from one half, we choose the bit complement of one of these, thereby handling all remaining triples. This gives us a covering array of size $N_2 + N_3$. ■

Chateauneuf *et al.* [3] prove a generalization, which we repeat here.

Theorem 2 $CAN(3, 2k, v) \leq CAN(3, k, v) + (v-1)CAN(2, k, v)$.

Proof. Begin as in Theorem 1 by placing two $CA(N_3; 3, k, v)$s side by side. Let C be a $CA(N_2; 2, k, v)$. Let π be a cyclic permutation of the v symbols. Then for $1 \leq i \leq v-1$, we append N_2 rows consisting of C and $\pi^i(C)$ placed side-by-side. The verification is as for Theorem 1. ■

We now develop a substantial generalization to permit the number of factors to be multiplied by $\ell \geq 2$ rather than two; this is the *k'ary Roux construction.* To carry this out, we require another combinatorial object. Let Γ be a group of order v, with $\odot$ as its binary operation. A *difference covering array* $D = (d_{ij})$ over Γ, denoted by $DCA(N, \Gamma; 2, k, v)$, is an $N \times k$ array with entries from Γ having the property that for any two distinct columns j and ℓ, $\{d_{ij} \odot d_{i\ell}^{-1} : 1 \leq i \leq N\}$ contains every *non-identity* element of Γ at least once. When Γ is abelian, additive notation is used, explaining the "difference" terminology. We shall only employ the case when $\Gamma = \mathbb{Z}_v$, and omit it from the notation. We denote by $DCAN(2, k, v)$ the minimum N for which a $DCA(N, \mathbb{Z}_v; 2, k, v)$ exists.

Theorem 3 $CAN(3, k\ell, v) \leq CAN(3, k, v) + CAN(3, \ell, v) + CAN(2, \ell, v) \times DCAN(2, k, v)$.

Proof. We suppose that the following all exist:

(1) a CA$(N; 3, \ell, v)$ A;
(2) a CA$(M; 3, k, v)$ B;
(3) a CA$(R; 2, \ell, v)$ F; and
(4) a DCA$(Q; 2, k, v)$ D.

We produce a CA$(N + M + QR; 3, k\ell, v)$ C (see Figure 1). For convenience, we index the $k\ell$ columns of C by ordered pairs from $\{1, \ldots, k\} \times \{1, \ldots, \ell\}$. C is formed by vertically juxtaposing three arrays, C_1 of size $N \times k\ell$, C_2 of size $M \times k\ell$, and C_3 of size $QR \times k\ell$. We describe the construction for each in turn.

C_1 is produced as follows. In row r and column (i, j) of C_1 we place the entry

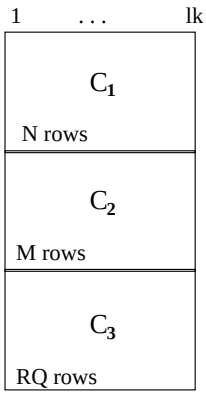

Fig. 1. k'ary Roux Construction

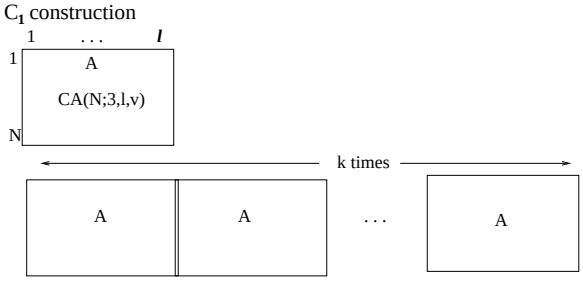

Fig. 2. Construction of C_1

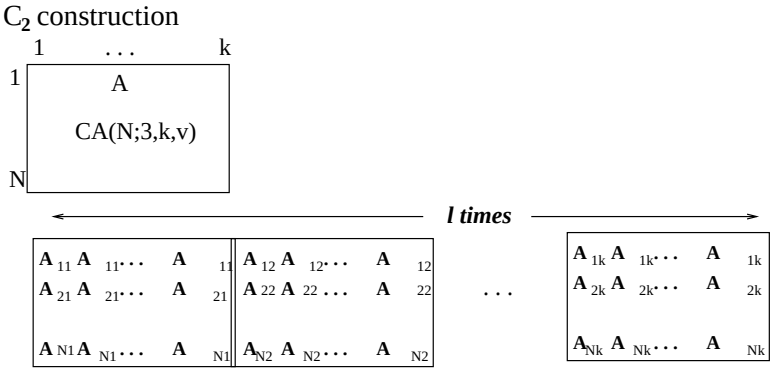

Fig. 3. Construction of C_2

in cell (r, j) of A. Thus C_1 consists of k copies of A placed side by side. This is illustrated in Figure 2.

C_2 is produced as follows. In row r and column (i, j) of C_2 we place the entry in cell (r, i) of B. Thus C_2 consists of ℓ copies of the first column of B, then ℓ copies of the second column, and so on (see Figure 3).

To construct C_3 (see Figure 4), let $D = (d_{ij} : i = 1, \ldots, Q; j = 1 \ldots, k)$ and $F = (f_{rs} : r = 1, \ldots, R; s = 1, \ldots, \ell)$. Choose a cyclic permutation π on the v symbols of the array. Then in row $(i-1)R + r$ and column (j, s) of C_3 place the entry $\pi^{d_{ij}}(f_{rs})$.

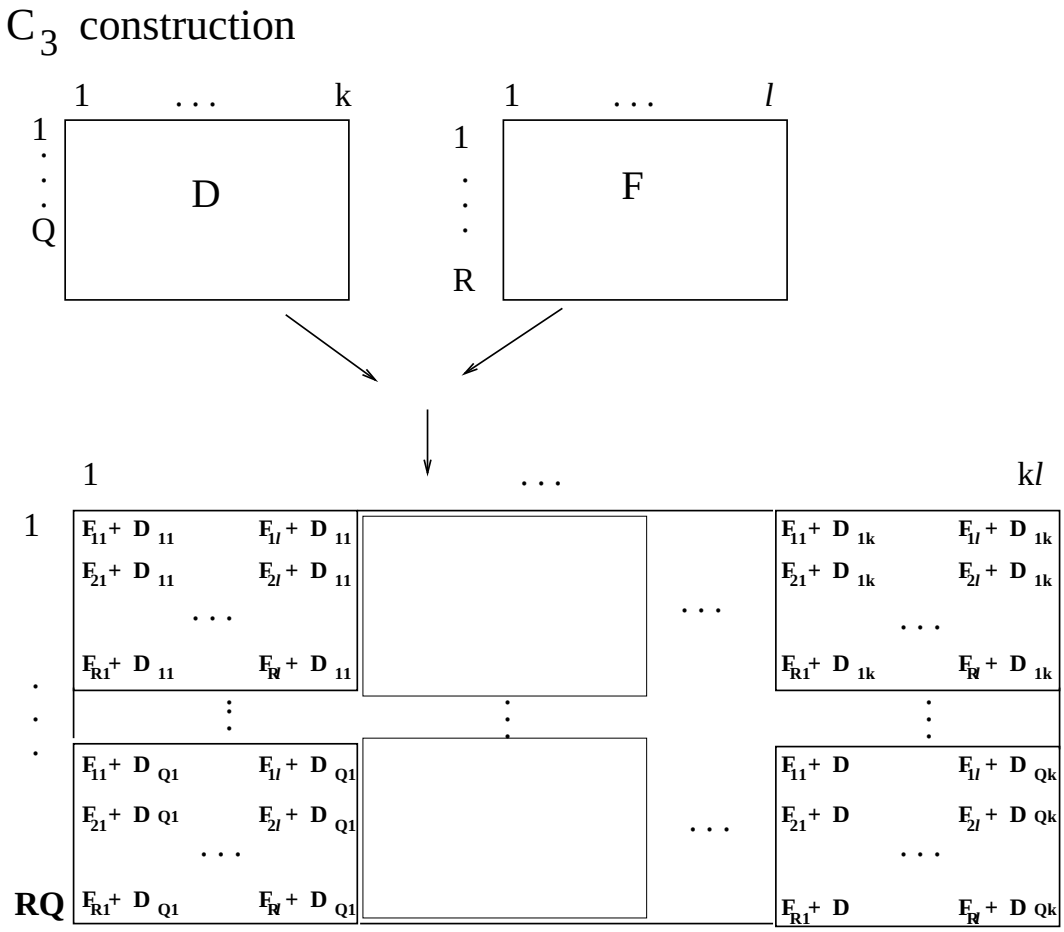

Fig. 4. Construction of C_3

We verify that C is indeed a CA$(N + M + QR; 3, k\ell, v)$. The only issue is to ensure that every 3 columns of C cover each of the v^3 3-tuples. Select three columns (i_1, j_1), (i_2, j_2), and (i_3, j_3) of C. If j_1, j_2 and j_3 are all distinct, then these three columns restricted to C_1 arise from three different columns of A, and hence all 3-tuples are covered. Similarly, if i_1, i_2, and i_3 are all distinct, then restricting the three columns to C_2, they arise from three distinct columns of B and hence again all 3-tuples are covered.

So we suppose without loss of generality that $i_1 = i_2 \neq i_3$ and $j_1 \neq j_2 = j_3$. The structure of C_3 consists of a $Q \times k$ block matrix in which each copy is a permuted version of F (under a permutation that is a power of π). That $i_1 = i_2$ indicates that two columns are selected from one column of this block matrix, and that i_3 is different means that the third column is selected from a different column of the block matrix. Now consider a selection $(\sigma_1, \sigma_2, \sigma_3)$ of symbols in the three chosen columns of C (actually, of C_3). Each selection of (σ_1, σ_2) appears in each block of the Q permuted versions of F appearing in the indicated column of the block matrix. Now suppose that $\sigma_3 = \pi^i(\sigma_2)$;

since π is a v-cycle, some power of pi satisfies this equality. Considering the permuted versions of F appearing in the columns corresponding to i_3, we observe that since D is an array covering all differences modulo v, in at least one row of the block matrix, we find that the block X in column i_3 and the block Y in column i_2 satisfy $Y = \pi^i(X)$. Hence every choice for σ_3 appears with the specified pair (σ_1, σ_2). ■

This can be improved upon: we do not need to cover triples when $\sigma_3 = \sigma_2$ since these are covered in C_1. Nor do we need to cover 3-tuples when $\sigma_1 = \sigma_2$, since these are covered in C_2. So we can eliminate some rows from F since we do not need to cover pairs whose symbols are equal in F. This modification improves further on the bounds.

3.3 Construction Using Generalized Hadamard Matrices

Augmented annealing affords the opportunity to develop "constructions" when some of the "ingredients" are not known at all. We illustrate this next. The basic plan is to simply construct a large portion of a covering array to use as a seed. Consider an $OA_\lambda(2, k, v)$. Each 2-tuple is covered exactly λ times. Some 3-tuples are also covered. Indeed, among the v 3-tuples containing a specified 2-tuple, at least one and at most $\min(v, \lambda)$ are covered. If λ of the v are covered for every 2-tuple, the orthogonal array is *supersimple*. Little is known about supersimple orthogonal arrays except when k is small. However our concern is only that "relatively many" triples are covered using "relatively few" rows. This is intentionally vague, since our intent is only to use the rows of the orthogonal array as a seed for a strength three covering array. A natural family of orthogonal arrays to consider arise from generalized Hadamard matrices (see [9]). We have no assurance that the resulting orthogonal arrays are supersimple, but instead choose generalized Hadamard matrices since they provide a means to cover many of the triples to be covered by the covering array. Although orthogonal arrays in general may be useful in constructions here, those from generalized Hadamard matrices appear frequently to cover either only one, or all v, of the triples containing a specified pair; this regularity appears to be beneficial. In the next section, we report computational results using these as seeds in annealing. The most important remark here is that, given such a generalized Hadamard matrix, it is not at all clear what "ingredients" are needed to complete it to a covering array in general, despite the fact that in any specific case we can easily enumerate the triples left uncovered.

0	0	0	0	0	0
1	1	1	1	1	1
0	1	1	0	1	0
0	0	1	1	0	1
0	0	0	0	1	1
0	1	0	1	1	1
1	1	0	1	0	0
1	1	0	0	0	1
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	0	1	0
0	1	1	1	0	0

Table 3
$TOCA(12; 3, 6, 2; 2)$

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

Table 4
$TOCA(13; 3, 10, 2; 2)$

4 Computational Results

4.1 Constructions Using Ordered Designs

We presented constructions for $CA(3, 6, 6)$s earlier. Construction 1 gave a size of 300, while Construction 2, requiring covering arrays with two disjoint rows, gave a size of 276. Table 3 gives a $CA(3, 6, 2)$ covering array with two disjoint rows.

We can create variations on this construction using augmented annealing. We can construct a $TOCA(30; 3, 6, 3; 0)$. The bound for a $CA(3, 6, 3)$ is 33 so we have saved three rows by using augmented annealing. We can use this to cover $\binom{3}{2} = 3$ combinations of the six pairs of symbols. There are still 12 remaining. We can cover these using 12 $TOCA(12; 3, 6, 2; 2)$s. Each of these are of size 10 once constant rows are removed. Lastly we add back in three rows of type a, a, a, a, a, a (we can exclude the three symbols covered by the $TOCA(3, 6, 3)$) and join these together. This gives us a covering array of size $120 + 30 + (12 \times 10) + 3 = 273$. This is smaller than the constructions given. Using instead two $TOCA(30; 3, 6, 3; 0)$s reduces the bound further to 270. Other linear spaces in Construction 3 can be employed. In the case of $CA(3, 6, 6)$ we found the best bound using only two building blocks. We used annealing to create an ordered design of size 120 and annealing to create a $TOCA(140; 3, 6, 6; 0)$. This gives us a $CA(260; 3, 6, 6)$, improving considerably on the constructions given above.

These applications of Construction 3 can be used in all of the cases outlined below. Tables 8-11 show the smallest sizes of (2,1)-covering arrays found by simulated annealing. The first column of each gives the size with v disjoint rows, and the second with no seeded rows.

Table 5 shows the smallest covering arrays found using two augmented methods and provides the smallest numbers we have obtained using straight an-

nealing as well as known bounds published in [3, 7]. The first method, labeled A, uses an ordered design combined with a $TOCA(3, k, k; 0)$ found by annealing. The second method uses an ordered design and combines it with $\binom{k}{2}$ $TOCA(3, k, 2; 2)$s. The best values we have found for these arrays are given in Table 6. The ordered design for $CA(3, 6, 6)$ was created using annealing. All of the other ordered designs were created using the definition of $PSL(2, q)$ (see [1]). Values in bold font are new upper bounds for these arrays.

In the case of $CA(3, 8, 8)$ and $CA(3, 9, 9)$, the collection of all triples can be covered exactly, i.e. every triple is covered precisely once (this is an orthogonal array of strength three). We therefore cannot improve over the best known result since it is optimal. However, these cases nevertheless illustrate improvement from augmented annealing over straight annealing. The smallest array we have found using simulated annealing in a reasonable amount of computational time for the $CA(3, 8, 8)$ has 918 rows. This result required almost three hours to run, illustrating the severity of the difficulty with naive computational search. We can instead create an $OD(3, 8, 8)$ of size 336 in significantly less time and anneal a (2,1)-covering array of size 280 in approximately five minutes. This provides us with a $CA(3, 8, 8)$ of size 616 which is smaller and computationally less expensive than using just annealing.

For $CA(3, 9, 9)$ similar results are found. In this case, however, using either $TOCA(3, 9, 2; 2)$s or a $TOCA(3, 9, 9; 0)$ does not fare as well as using Construction 3 with a linear space consisting of twelve blocks of size three; then $CAN(3, 9, 9) \leq 900$ is obtained. Perhaps this serves well to illustrate a general conclusion. An optimal solution has 729 rows, while annealing alone takes substantial time to obtain a bound of 1490. Augmented annealing yields a bound of 900 quickly, and applies more generally than the existence of an orthogonal array.

For the $CA(3, 10, 10)$ we can use the ordered design construction to generate the first part of this array. We can build 45 $TOCA(13; 3, 10, 2; 2)$s and add back in 10 rows of type $a, a, .., a$. If we do this we have an array of size 1225 which improves upon the published bound of 1331 [3]. We can also build a $TOCA(499; 3, 10, 10; 0)$ using annealing. When combined with the ordered design, the size of the covering array is 1219. The smallest array we have built with straight annealing for a $CA(3, 10, 10)$ is of size 2163. Again using Construction 3 with a suitably chosen linear space yields the best known result. A linear space with three lines of size four and nine of size three gives $CAN(3, 10, 10) \leq 1215$. It appears that the $TOCA(3, 10, 10; 0)$ is not yielding as strong a result in part because it has, in some sense, become a "large" ingredient and annealing is not as effective.

$CA(t,k,v)$	Augmented Annealing		Simulated Annealing	Smallest Reported[1] Array Size
	A	B		
$CA(3,6,6)$	**260**	276	300	300
$CA(3,8,8)$	616	624	918	512
$CA(3,9,9)$	906	909	1,490	729
$CA(3,10,10)$	**1,219**	1,225	2,163	1,331
$CA(3,12,12)$	2,339	**2,190**	4,422	2,197
$CA(3,14,14)$	4,134	**3,654**	8,092	4,096

Table 5
Sizes for covering arrays using augmented annealing.
Method A = $TOCA(3,k,k;0)$, Method B = $TOCA(3,k,2;2)$s
1. Source = Chateauneuf *et al.*[3] and Cohen *et al.* [7]

$TOCA(3,q+1,2;2)$s t,k,v	Size
$3,6,2$	12
$3,8,2$	12
$3,9,2$	13
$3,10,2$	13
$3,12,2$	15
$3,14,2$	18

Table 6
Sizes for $TOCA(3,q+1,2;2)$s

$TOCA(3,q+1,q+1;0)$ t,k,v	Size	Ordered Design
$3,6,6$	140	120
$3,8,8$	280	336
$3,9,9$	402	504
$3,10,10$	499	720
$3,12,12$	1,019	1,320
$3,14,14$	1,950	2,184

Table 7
Sizes for $TOCA(3,q+1,q+1;0)$s and ordered designs

t, k, v	$TOCAN(t, k, v; v) \leq$	$TOCAN(t, k, v; 0) \leq$
$3, 6, 2$	12	12
$3, 6, 3$	33	30
$3, 6, 4$	60	56
$3, 6, 5$	99	94
$3, 6, 6$	145	140

Table 8
Sizes for TOCAs with $k = 6$

t, k, v	$TOCAN(t, k, v; v) \leq$	$TOCAN(t, k, v; 0) \leq$
$3, 8, 2$	12	12
$3, 8, 3$	33	30
$3, 8, 4$	64	60
$3, 8, 5$	105	100
$3, 8, 6$	156	150
$3, 8, 7$	217	210
$3, 8, 8$	288	280

Table 9
Sizes for TOCAs with $k = 8$

t, k, v	$TOCAN(t, k, v; v) \leq$	$TOCAN(t, k, v; 0) \leq$
$3, 9, 2$	13	12
$3, 9, 3$	36	33
$3, 9, 4$	70	67
$3, 9, 5$	116	110
$3, 9, 6$	171	166
$3, 9, 7$	239	233
$3, 9, 8$	316	308
$3, 9, 9$	416	402

Table 10
Sizes for TOCAs with $k = 9$

4.2 Constructions from Generalized Hadamard Matrices

Table 12 gives some results for building covering arrays from strength two orthogonal arrays of index higher than one. These are from Generalized Hadamard

t, k, v	$TOCAN(t, k, v; v) \leq$	$TOCAN(t, k, v; 0) \leq$
$3, 10, 2$	13	12
$3, 10, 3$	36	33
$3, 10, 4$	70	66
$3, 10, 5$	115	111
$3, 10, 6$	172	165
$3, 10, 7$	239	232
$3, 10, 8$	322	310
$3, 10, 9$	409	401
$3, 10, 10$	506	499

Table 11
Sizes for TOCAs with $k = 10$

CA	Size	Previous Bound*	OA,Size	Percent triples covered by OA
$CA(3, 9, 3)$	**50**	51	$OA_3(2, 9, 3)$,27	90.5
$CA(3, 25, 5)$	**371**	465	$OA_5(2, 25, 5)$,125	89.6
$CA(3, 27, 3)$	118	99	$OA_9(2, 27, 3)$,81	97.3
$CA(3, 16, 4)$	174	159	$OA_4(2, 16, 4)$,64	78.6
$CA(3, 17, 4)$	**180**	184	$OA_4(2, 17, 4)$,64	77.9
$CA(3, 10, 5)$	266	185	$OA_2(2, 10, 5)$,50	40.0

Table 12
Sizes for CAs built with OAs of higher index *Source: [3]

$CA(t, k\ell, v)$	Size	Previous Bound*	$CA(3, k, v)$,Size	$CA(3, \ell, v)$,Size	Size of D	Size of F
$CA(3, 25, 4)$	**188**	229	$CA(3, 5, 4)$,64	$CA(3, 5, 4)$,64	4	15
$CA(3, 30, 4)$	**203**	238	$CA(3, 5, 4)$,64	$CA(3, 6, 4)$,64	5	15
$CA(3, 24, 6)$	**692**	795	$CA(3, 6, 6)$, 260	$CA(3, 4, 6)$,216	6	36
$CA(3, 36, 3)$	109	?	$CA(3, 4, 3)$,27	$CA(3, 9, 3)$,50	3	8

Table 13
K'ary Roux. * Source: [3]

Matrices. In each of these cases we seed the annealing program with the OA of higher index and then anneal the rest of the array. We include in the table the percentage of triples covered by and the size of the OA prior to annealing. In our experience if too many triples are covered before annealing occurs, the best bound is not found. There seems to be a tradeoff in the tightness of the structure used for seeding and the final covering array. We have listed the size of the strength two orthogonal array and the percentage of triples that are covered in this subset.

4.3 Constructions Using k'ary Roux

We applied the k'ary Roux construction to some covering arrays for sizes of $\ell > 2$. Table 13 gives some of these results. This construction appears to do well when the two smaller building blocks are themselves optimal. In the first two entries we have used orthogonal arrays as ingredients. In each of these entries we do not have to handle triples when $\sigma_1 = \sigma_2$. We have used the augmented annealing program to build D and F by initializing them with these triples. The sizes we found for these are listed in the table. For instance, we can build a difference covering array of size $DCA(4; 2, 5, 4)$ of size 4 instead of 5 if we do not care about covering the zero differences. And we can create a $CA(2, 6, 4)$ of size 15 if we do not care about pairs with equal entries. This saves us 15 rows in the final covering array.

Table 14 gives results of computations using simulated annealing for the existence of difference covering arrays. When two entries are given, the first is for a DCA that (in addition) covers the zero difference, while the second does not require the zero difference to be covered.

4.4 Arrays with No Known Algebraic Constructions

We close with some examples in which no combinatorial construction is available for one or more ingredients. The first example is a $CA(3, 7, 7)$. We have used annealing to create (2,1)-covering arrays and analogs of ordered designs. We only improve slightly on the best bound found for this array from straight annealing, but appear to improve on the computation time that is required to solve this problem. The second example is an $MCA(3, 6^6 4^2 2^2)$. This array contains a $CA(3, 6, 6)$ but has four additional components. We have tried several techniques to build this array. When we use straight annealing we found an array of size 317, which is much larger than the best bound we have found for the sub-array $CA(3, 6, 6)$. Based on the experience reported in [6] we believe that the hardest problem, that of the $CA(3, 6, 6)$, dictates the size of this array. When we used two partial covering arrays as in Method B, the best bound

Sizes for DCA's with,without Zero Differences								
k/q	3	4	5	6	7	8	9	10
3	3,2	5,4	5,4	7,6	7,6	9,8	9,8	11,10
4	4,3	5,4	5,4	7,6	7,6	9,8	10,9	11,10
5	5,4	5,4	5,4	8,7	7,6	9,8	10,9	12,11
6	5,4	6,5	7,6	8,7	7,6	10,9	11,10	12,11
7	5,4	6,5	7,6	8,7	7,6	10,9	11,11	13,12
8	5,4	6,5	8,7	8,7	9,8	10,9	12,11	14,13
9	5,4	7,6	8,7	9,8	9,8	12,11	12,12	14,13
10	5,4	7,6	8,7	9,8	10,9	12,11	13,12	15,14

Table 14
Table of Difference Covering Arrays with,without zero differences

t, k, v	$TOCAN(t, k, v; v) \leq$	$TOCAN(t, k, v; 0) \leq$
$3, 7, 2$	12	12
$3, 7, 3$	33	30
$3, 7, 4$	64	60
$3, 7, 5$	105	100
$3, 7, 6$	156	150
$3, 7, 7$	217	210

Table 15
Sizes for $TOCA(3, k, v)$s with $k = 7$

we found was 313. We have therefore tried seeding this array with solutions for subproblems already found. We seed either the $OD(3, 6, 6)$ of size 120 or the $CA(3, 6, 6)$ of size 263 and then anneal to complete the structure. Both of these improve markedly upon the first two methods as shown in Table 16. The smallest test suite we found used the $CA(3, 6, 6)$ as a seed. This added fewer than 10 rows to complete the missing coverage. This highlights the need for the software tester to have knowledge to determine which method is best for which problem.

Covering Array	Method	Size
$CA(3, 7, 7)$	Straight Annealing	552
$CA(3, 7, 7)$	Partial Arrays	545
$MCA(3, 6^6 4^2 2^2)$	Straight Annealing	317
$MCA(3, 6^6 4^2 2^2)$	Partial Arrays	313
$MCA(3, 6^6 4^2 2^2)$	Seeded with $OD(3, 6, 6)$	283
$MCA(3, 10, 6^6 4^2 2^2)$	Seeded with $CA(3, 6, 6)$	272

Table 16
Sizes for covering arrays with no known combinatorial constructions

5 Conclusions

The construction of covering arrays is a challenging combinatorial and computational problem. Their real and potential applications in the design of software test suites necessitate reasonably fast and reasonably accurate techniques for producing large covering arrays. Computational search techniques, while general, degrade in speed and accuracy as problem size increases. Combinatorial techniques suffer lack of generality despite offering the promise of fast and accurate solutions in specfic instances. We have therefore proposed a framework for combining combinatorial constructions with heuristic search, and examined a specific instantiation of this, augmented annealing. The covering arrays produced illustrate the potential of this approach, demonstrating that a combinatorial construction can be used as a master to decompose a search problem so that much smaller ingredient designs can be found. Perhaps what distinguishes this from the majority of existing recursive constructions is that we are not concerned primarily with finding a master for which the ingredients needed are themselves well-understood combinatorial objects. Augmented annealing can be viewed as a first step in designing a tool to exploit combinatorial constructions along with heuristic search to produce covering arrays for the variety of parameters arising in practice.

Acknowledgments

Research is supported by the Consortium for Embedded and Internetworking Technologies and by ARO grant DAAD 19-1-01-0406. Thanks to Peter B. Gibbons, Rick B. Mugridge and James S. Collofello for helpful comments and to the Consortium for Embedded and Internetworking Technologies for making a visit to ASU possible.

References

[1] P.J. Cameron. *Notes on Classical Groups.* Queen Mary, University of London, School of Mathematical Sciences, 2000. http://www.maths.qmw.ac.uk/ pjc/books.html

[2] M.A. Chateauneuf, C.J. Colbourn, and D.L. Kreher, Covering arrays of strength three, *Designs, Codes and Cryptography* 16 (1999), 235-242.

[3] M. Chateauneuf and D. Kreher. On the state of strength-three covering arrays. *Journal of Combinatorial Designs*, 10(4):217–238, 2002

[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):437–44, 1997.

[5] D. M. Cohen and M. L. Fredman. New techniques for designing qualitatively independent systems. *Journal of Combinatorial Designs*, 6(6):411–16, 1998.

[6] M. B. Cohen, C. J. Colbourn, J.S. Collofello, P. B. Gibbons and W. B. Mugridge. Variable Strength Interaction Testing of Components. Proc. of 27th Intl. Computer Software and Applications Conference (COMPSAC 2003) to appear.

[7] M. B. Cohen, C. J. Colbourn, P. B. Gibbons and W. B. Mugridge. Constructing test suites for interaction testing. In *Proc. of the Intl. Conf. on Software Engineering (ICSE 2003)*, Portland, Oregon, May 2003, pp 38-49.

[8] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, Augmented simulated annealing to build interaction test suites, preprint, 2003.

[9] C. J. Colbourn and W. De Launey, Difference Matrices, in [10], pp. 287–297.

[10] C. J. Colbourn and J. H. Dinitz (editors), *The CRC Handbook of Combinatorial Designs*, CRC Press, Boca Raton, 1996.

[11] C. Colbourn and J. Dinitz. Making the MOLS table. In *Computational and Constructive Design Theory*, 1996. (W.D.Wallis, ed.) Kluwer Academic Press, 67-134.

[12] S. R. Dalal, A. J. N. Karunanithi, J. M. L. Leaton, G. C. P. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proc. of the Intl. Conf. on Software Engineering,(ICSE '99)*, 1999, pp. 285-94, New York.

[13] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. In *Proc. of the Intl. Conf. on Software Engineering, (ICSE '97)*, 1997, pp. 205-215, New York.

[14] A. Hedayat, N. Sloane, and J. Stufken. *Orthogonal Arrays.* Springer-Verlag, New York, 1999.

[15] K. Nurmela and P. R. J. Östergård. Constructing covering designs by simulated annealing. Technical report, Digital Systems Laboratory, Helsinki Univ. of Technology, 1993.

[16] N. Sloane. Covering arrays and intersecting codes. *Journal of Combina-*

torial Designs, 1(1):51–63, 1993.

[17] B. Stevens and E. Mendelsohn. New recursive methods for transversal covers. *Journal of Combinatorial Designs*, 7(3):185–203, 1999.

[18] B. Stevens, L. Moura, and E. Mendelsohn. Lower bounds for transversal covers. *Designs Codes and Cryptography*, 15(3):279–299, 1998.

[19] D. R. Stinson. *Cryptography, Theory and Practice.* CRC Press, Boca Raton, FL, 1995.

[20] K. C. Tai and L. Yu. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28(1):109-111, 2002.

[21] A. W. Williams and R. L. Probert. A practical strategy for testing pairwise coverage of network interfaces. In *Proc. Seventh Intl. Symp. on Software Reliability Engineering*, 1996, pp. 246-54.

[22] A. W. Williams and R. L. Probert. A measure for component interaction test coverage. In *Proc. ACS/IEEE Intl. Conf. on Computer Systems and Applications*, 2001, pp. 301-311.

[23] T. Yu-Wen and W. S. Aldiwan. Automating test case generation for the new generation mission software system. In *Proc. IEEE Aerospace Conf.*, 2000, pp. 431-437.