# Constructing Test Suites for Interaction Testing

Myra B. Cohen
Peter B. Gibbons
Warwick B. Mugridge
Dept. of Computer Science
University of Auckland
Private Bag 92019
Auckland, New Zealand
{myra,peter-g,rick}@cs.auckland.ac.nz

Charles J. Colbourn
Dept of Computer Science and Engineering
Arizona State University
P.O. Box 875406
Tempe, Arizona 85287
charles.colbourn@asu.edu

## Abstract

*Software system faults are often caused by unexpected interactions among components. Yet the size of a test suite required to test all possible combinations of interactions can be prohibitive in even a moderately sized project. Instead, we may use pairwise or $t$-way testing to provide a guarantee that all pairs or $t$-way combinations of components are tested together. This concept draws on methods used in statistical testing for manufacturing and has been extended to software system testing. A covering array, $CA(N; t, k, v)$, is an $N \times k$ array on $v$ symbols such that every $N \times t$ sub-array contains all ordered subsets from $v$ symbols of size $t$ at least once. The properties of these objects, however, do not necessarily satisfy real software testing needs. Instead we examine a less studied object, the mixed level covering array and propose a new object, the variable strength covering array, which provides a more robust environment for software interaction testing. Initial results are presented suggesting that heuristic search techniques are more effective than some of the known greedy methods for finding smaller sized test suites. We present a discussion of an integrated approach for finding covering arrays and discuss how application of these techniques can be used to construct variable strength arrays.*

## 1. Introduction

Many software systems today are built using components. Often, system faults are caused by unexpected interactions among these [28]. For example, suppose we have an Internet-based software system. Our customers may use a variety of browsers. In addition they may be using different operating systems, connection types and printer configurations. In order to completely test this system we want to test our software on all of the possible supported configurations. If we have the system shown in Table 1, we would want to test combinations, such as (Netscape,Windows,LAN,Local) and (Netscape,Windows,LAN,Networked). To test all possible interactions for this system we would need $3^4$ or 81 configurations.

| Component | | | |
|---|---|---|---|
| **Web Browser** | **Operating System** | **Connection Type** | **Printer Config** |
| Netscape | Windows | LAN | Local |
| IE | Macintosh | PPP | Networked |
| Other | Linux | ISDN | Screen |

**Table 1. Four Components, Each With Three Configurations**

This may be reasonable for a small system but the number of necessary tests grows large very quickly. Suppose we had 10 possible components with four possible settings each. We then need $4^{10}$=1,048,576 test configurations. One approach used is to guarantee that we test all pairs of interactions or all *n*-way interactions [2, 5, 17, 12, 23, 27, 29]. In the example shown in Table 1 we can cover all pairs of interactions using only nine different configurations (see Table 2). And in the example of 10 components each with 4 possible settings we can cover all pairs of interactions using at most 25 configurations.

Dalal et al. present empirical results to argue that the testing of all pairwise interactions in a software system finds a large percentage of the existing faults [10]. In further work, Burr et al. provide more empirical results to show that this type of test coverage is effective [3].

If we restrict ourselves to pairwise coverage, we cannot guarantee that we will find faults that occur with three or

| Test | Browser | OS | Connection | Printer |
|---|---|---|---|---|
| 1 | NetScape | Windows | LAN | Local |
| 2 | NetScape | Linux | ISDN | Networked |
| 3 | NetScape | Macintosh | PPP | Screen |
| 4 | IE | Windows | ISDN | Screen |
| 5 | IE | Macintosh | LAN | Networked |
| 6 | IE | Linux | PPP | Local |
| 7 | Other | Windows | PPP | Networked |
| 8 | Other | Linux | LAN | Screen |
| 9 | Other | Macintosh | ISDN | Local |

**Table 2. Test Suite to Cover all Pairs from Table 1**

four way interactions. A trade off occurs between the time and cost of testing and the required degree of guaranteed coverage. Williams et al. describe a method to quantify the coverage for a particular interaction level [28]. We can determine how many pairs, or $n$-way interactions we have covered at each stage when building a test suite. For instance if we have four components, any new test case can contribute at most $\binom{4}{2}$, or 6 new covered pairs. Further, if each component has three configurations, there are a total of $\binom{4}{2}\ 3^2 = 54$ possible pairs that must be covered. Therefore any one new test case increases our coverage by at most $11.1\%$ [28]. A similar method is described by Dunietz et al. [12].

It could be assumed that our goal is for $100\%$ coverage at a particular level, such as two or three way coverage. This can be achieved using a minimal number of test cases, with a known combinatorial object, the *covering array*, described below. Dunietz et al. [12] point out that by using such a balanced code, we are also providing higher coverage for stronger interaction levels.

As it is usually too expensive to test all components using three or four way coverage we can benefit from doing this for part of the system. For instance, a particular subset of components may have a higher interaction dependency or a certain combination of components may have more serious effects in the event of a failure. For example, consider a subset of components that control a safety-critical hardware interface. We want to use stronger coverage in that area. However, the rest of our components may be sufficiently tested using pair-wise interaction. We can assign a coverage *strength* requirement to each subset of components as well as to the whole system.

For the safety critical system, we would require that the whole system has $100\%$ coverage for two way interactions, while the safety-critical subset has 100% coverage for three way interactions. The final test suite may, however, have 80% coverage for three way interactions over all components.

In the rest of this paper we examine the combinatorial objects that can be used in component interaction testing. We begin with a description of relevant combinatorial objects and introduce a new model for variable strength interaction testing. We discuss various techniques to build these test suites and present preliminary results using heuristic search techniques. Lastly we discuss how the techniques can be applied and used to build our new model for variable strength interaction.

## 2. Combinatorial Models

Combinatorial objects are not new to testing. Hedayat et al. [14] discuss the use of orthogonal arrays for statistically designed experiments. These are used across many disciplines including medicine, agriculture and manufacturing [14]. More recently these ideas have been extended to software testing.

### 2.1. Orthogonal Latin Squares

Mandl proposed using orthogonal Latin squares for testing compilers [17]. A Latin square of order $s$ is an $s \times s$ array with entries from a set $S$ of cardinality $s$ with the condition that for all $i$ in $S$, $i$ appears exactly once in each row and each column. Two Latin Squares are orthogonal if, when superimposed on each other, the ordered pairs created in each cell cover all $s^2$ pairs [1, 14].

In this instance, $k$ orthogonal Latin squares of size $s$ are needed to test $k+2$ parameters each with $s$ levels. The cell entries represent $k-2$ elements in each test, and the column and row indices represent the additional 2 parameters.

### 2.2. Orthogonal Arrays

Brownlie et al. have adapted an engineering concept, called *Robust Testing*, to the task of testing software [2]. They developed the OATs system which uses orthogonal arrays to generate test suites for a software system [2]. $k$ mutually orthogonal Latin squares of order $s$ can be transformed into an orthogonal array $OA_1(s^2; k+2, s, 2)$ [1].

An orthogonal array $OA_\lambda(N; k, v, t)$ 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* $\lambda$ times [14]. Orthogonal arrays have the property that $\lambda = \frac{N}{v^t}$. Table 2 is an example of an $OA(9; 4, 3, 2)$.

Although the use of orthogonal arrays for testing has been discussed in the literature [2, 27] these may be of less interest in component testing because they could lead to overly large test suites with $\lambda > 1$. For cases of $v$ and $k$ where an orthogonal array with $\lambda = 1$ does exist, clearly this is the optimal test suite. However, there are many values of $v$ and $k$ where an orthogonal array with $\lambda = 1$ does

not exist so we must resort to a less restrictive structure; one that requires subsets are instead covered *at least* once as with covering arrays.

### 2.3. Covering Arrays

A *covering array*, $CA_\lambda(N; t, k, v)$, is an $N \times k$ array on $v$ symbols such that every $N \times t$ sub-array contains all ordered subsets from $v$ symbols of size $t$ *at least* $\lambda$ times. When $\lambda = 1$ we use the notation $CA(N; t, k, v)$. In such an array, $t$ is called the *strength*, $k$ the *degree* and $v$ the *order*. A covering array is optimal if it contains the minimum possible number of rows. We call the minimum number the *covering array number*, $CAN(t, k, v)$. For example, $CAN(2, 5, 3) = 11$ [4, 21].

Cohen et al. [5, 7] have developed the commercial product, *Automatic Efficient Test Generator* (AETG) to construct covering arrays for software interaction testing. Williams et al. [27] use orthogonal arrays as well as covering arrays to design tests for the interactions of nodes in a network. Stevens et al. [23] suggest creating a knowledge system for the tester that contains the best known covering arrays applicable to testing.

There are often a number of different ways to represent the same combinatorial object. For instance, several other combinatorial objects have been defined with the same effective properties as a covering array. A *strength t transversal cover*, a *qualitatively independent* system and *t-surjective array* [8, 21, 25] are three such objects. In the rest of this document we will use the covering array definition for consistency.

Covering arrays only suit the needs of software testers when all components have the same number of configurations. However, this is often not the case. For instance one component can have four possible configurations and one only two. Indeed, this is a normal occurrence. In addition, constraints can exist for a test suite suggesting that certain value combinations can never occur or that there are aggregate conditions among several fields [10]. Lastly, testers often require that a set of fixed test cases are added to each test suite, regardless of the interaction strength required. These issues cause real testing environments to deviate from covering arrays as defined. In the first case we want to remove these combinations from the tests. In the second case we may need to define aggregate fields or components. In the last case we need to build a test suite on top of already generated test cases. We restrict our subsequent discussion to the first problem, which we feel is the most important deviation from a *fixed level* covering array; however the techniques we discuss could be extended to the others.

Mixed Level Covering Array

| | | | |
|---|---|---|---|
| 0 | a | 4 | d |
| 2 | b | 6 | e |
| 3 | c | 5 | e |
| 2 | c | 4 | d |
| 0 | b | 5 | d |
| 1 | a | 6 | e |
| 1 | b | 4 | d |
| 3 | a | 6 | d |
| 0 | c | 6 | e |
| 2 | a | 5 | e |
| 3 | b | 4 | e |
| 1 | c | 5 | d |

**Figure 1.** $MCA(12; 2, 4^1 3^2 2^1)$

### 2.4. Mixed Level Covering Arrays

The variation among component levels can be handled with the *mixed level covering array*. Several authors have suggested its use for software testing (see [4, 22, 28]), but few results are known about upper bounds and how to construct these.

A *mixed level* covering array,

$$MCA_\lambda(N; t, k, (v_1 v_2 ... v_k)),$$

is an $N \times k$ array on $v$ symbols, where $v = \sum_{i=1}^{k} v_i$, with the following properties:

1. Each column $i \quad (1 \leq i \leq k)$ contains only elements from a set $S_i$ of size $v_i$.

2. The rows of each $N \times t$ subarray cover all $t$-tuples of values from the $t$ columns at least $\lambda$ times.

When $\lambda = 1$ we can omit the subscript, representing the array as $MCA(N; t, k, (v_1 v_2 ... v_k))$. We use a shorthand notation to describe mixed covering arrays by combining equal entries in $(v_i : i \leq 1 \leq k)$. For example three entries each equal to 2 can be written as $2^3$. Consider an $MCA(N; t, (w_1^{k_1} w_2^{k_2} ... w_s^{k_s}))$. This can also be written as an $MCA(N; t, k, (v_1 v_2 ... v_k))$ (see Figure 1). In this array we have:

1. $k = \sum_{i=1}^{s} k_i \quad$ and $\quad v = \sum_{i=1}^{s} k_i w_i = \sum_{i=1}^{k} v_i$.

2. Each column $i \quad (1 \leq i \leq k)$ contains only elements from a set $S_i$ where $|\cup_{i=1}^{k} S_i| = v$.

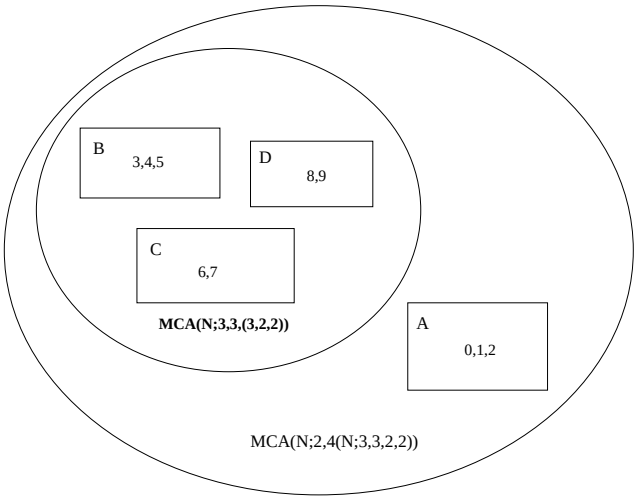


**Figure 2. Model of Variable Strength Covering Array.**

| Component | | | |
|---|---|---|---|
| **A** | **B** | **C** | **D** |
| 0 | 4 | 7 | 8 |
| 1 | 3 | 7 | 9 |
| 2 | 3 | 6 | 8 |
| 0 | 5 | 6 | 9 |
| 1 | 4 | 6 | 8 |
| 2 | 4 | 7 | 9 |
| 2 | 5 | 7 | 9 |
| 1 | 5 | 7 | 8 |
| 0 | 3 | 6 | 9 |
| | | | |
| 0 | 3 | 7 | 8 |
| 2 | 4 | 6 | 9 |
| 1 | 5 | 6 | 8 |

**Table 3. Variable Strength Array for Figure 2**

3. The columns are partitioned into $s$ groups $g_1, g_2, \ldots g_s$ where group $g_i$ contains $k_i$ columns. The first $k_1$ columns belong to the group $g_1$, the next $k_2$ columns belong to group $g_2$, and so on.
4. If column $r \in g_i$, then $|S_r| = v_i$.

This notation can be used for a fixed-level covering array as well. $CA(N; t, v^k)$ indicates that there are $k$ parameters each containing a set of $v$ symbols. This makes it easier to see that the values from different components can come from different sets.

### 2.5. Variable Strength Covering Arrays

Being able to guarantee that subsets of components have higher interaction strength is appealing in a real test environment as with the safety-critical example in Section 1. A covering array as it is defined only guarantees a coverage of $t$-subsets. We may also wish to have coverage of some subsets of size $t'$ for values of $t' > t$. It is for this reason that we present a new model for interaction testing, as illustrated in Figure 2.

Components A and B have three configurations each, while C and D have two (the configurations are labelled $0, 1, 2 \ldots 9$). The system has a total of 37 interaction pairs, and 60 interaction triples. We require that the subset $\{B, C, D\}$ be covered by 3-way coverage, while the entire system $\{A, B, C, D\}$ is to have 2-way coverage. We therefore require a 100% 2-way coverage and a minimal 3-way coverage of $\frac{12}{60} = 20\%$. As we shall see, the actual 3-way coverage of a test suite may be much higher.

A mixed level covering array to handle $100\%$ of 2-way interactions can be created using only 9 tests. This is shown as the first 9 rows of Table 3. To handle all 3-way interactions, as many as 18 tests are needed. To increase the strength of this array to cover all 3-way interactions of $\{B, C, D\}$, we only need to add 3 more tests. These are the last three rows shown in Table 3. This actually covers $\frac{42}{60} = 70\%$ of the 3-way interactions for our system while covering all of the 3-way interactions for our subset and all 2-way interactions in the whole system.

## 3. Constructing Covering Arrays

In this section we begin with some known results and present several techniques for finding covering arrays.

### 3.1. Known Combinatorial Results

There are several types of results known for covering arrays. These include probabilistic bounds that provide a value for the smallest size of $N$, but do not give us any method for construction. There are constructive results which provide a direct way to create such an object, and finally, there are computational results which are produced as the end product of a search. Of these, the last two are probably the most useful, although knowing the probabilistic bounds helps to guide the search for new constructions. We give a few bounds below, but leave the readers to see Sloane [21] for an excellent survey.

The first known results on covering array numbers are due to Rényi [20] who determined these numbers for the case $t = v = 2$ when $N$ is even. Kleitman and Spencer [16] and Katona [15] independently determined covering array numbers for all $N$ (and $t = v = 2$). They showed that the

size of N grows as follows:

$$k = \binom{N-1}{\lceil \frac{N}{2} \rceil}$$

For a large $k$, this grows logarithmically. In 1990 Gargano, Körner and Vaccaro [13] gave the following probabilistic bound for $t = 2$, $v > 2$:

$$N = \frac{v}{2} log k(1 + o(1))$$

Östergård [19] showed that $N \leq 11$ for $v = 3, k = 5$, while Sloane [21] mentions that Applegate showed $N = 11$ [21]. More work has been done on these smaller cases. When $t = 3$, the combinatorial research illustrates both the depth of the connection with combinatorial configurations and the difficulties that these pose for software testers. We do not attempt a thorough review here, but instead refer the reader to [4].

### 3.2. Algebraic Constructions

We give a brief outline of an algebraic construction for a covering array to provide a flavour of constructive techniques. (For a detailed description see [24]). To begin with, we add another condition to our definition of a covering array; the requirement that there are $n$ disjoint rows, i.e. each pair has no $t$-sets in common.

We begin with $k - 2$ incomplete Mutually Orthogonal Latin Squares (MOLS) of order $n$ that have holes of size $b_1, b_2, ...b_s$. We construct our array by *filling* the holes with covers of order $b_i$ and degree $k$. We do this by using a set of bijections between the groups of the MOLS and the groups of the $CA(t, k, b_i)$.

Although this provides us with an array with a good known upper bound, this is not always easy to construct. First of all it requires that we find $k - 2$ incomplete MOLS of order $n$. We may be able to find some of this information in known tables, but in general it is not an easy task (see [9]). Secondly, it is only useful for certain values of $n, t$, and $k$.

What is striking from our viewpoint is that, 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. In addition, the true challenge facing the software tester is to determine when the construction applies, including what auxiliary ingredients are needed. This overhead limits the applicability of the more complex constructions.

### 3.3. Heuristic Search Techniques

Computational search techniques to find fixed level covering arrays include standard techniques such as hill climbing and simulated annealing. However, no results have been produced for mixed level arrays using these methods. The only approaches that produce mixed level arrays use greedy methods to find test suites [5, 29, 30]. Initial results shown in Section 5 suggest that combinatorial search techniques have the potential to produce smaller test suites.

There are two problems to address when building covering arrays. One is that we must satisfy the constraint of covering all $t$-sets of interactions. The other is that we do not know how large our final test suite will be. Test cases can be added incrementally, stopping when the constraints are met, which is the greedy approach. Alternatively we can choose a fixed size array and try to manipulate the space until all constraints are met. This is the method used by the other algorithms.

Other heuristic search techniques for covering arrays have been explored, specifically tabu search and genetic algorithms [22]. We omit them from our discussion here, since Stardom's research indicates that they are not as effective in general as simulated annealing. However, we present one further variant of simulated annealing, the great deluge algorithm (Section 3.7).

### 3.4. Greedy Algorithms

The AETG system and the Test Case Generator (TCG) [5, 7, 30] use a greedy search technique. Each test suite is built one test at a time, i.e. an $N \times k$ array is built one row at a time. For each subsequent test case to be added, many are created and then the best chosen (see [5, 7, 30]). The greedy portion of these algorithms lies in the step of determining which new symbol to add to each column of each test. This is of course a local optimum.

In each algorithm, information is maintained about which test case interactions are still uncovered and is used as a heuristic to provide a better chance of finding the missing interactions. AETG uses a random approach to finding a pool of test cases. Yu-Wen et al. [30] suggest a deterministic algorithm. The authors begin with a deterministic ordering of the parameters. Another greedy algorithm, the In Parameter-Order (IPO), has the benefit of reusing old test cases when new parameters are added. It does this by expanding in a vertical and horizontal fashion [29].

TCG and IPO have only been tested on pairwise interactions. AETG and TCG seem to be the most effective in finding pairwise interactions, but do not always produce test suites with known covering array numbers.

### 3.5. Hill Climbing

Hill climbing and simulated annealing are variants of the state space search technique for solving combinatorial optimisation problems. With a general optimisation problem the hope is that the found solution is close to an optimal one.

With many design problems we *know* (from the cost) when we have reached an optimal solution. On the other hand, approximations in these cases are of little value.

An optimisation problem can be specified as a set $\Sigma$ of feasible solutions (or states) together with a cost $c(S)$ associated with each $S \in \Sigma$. An optimal solution corresponds to a feasible solution with overall (i.e. global) minimum cost. We define, for each $S \in \Sigma$, 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 called the neighbourhood $N(S)$ of $S$.

We start by randomly choosing an initial feasible solution and then generate a randomly chosen transformation of the current feasible solution $S$. If the transformation results in a feasible solution $S'$ of equal or lower cost, then $S'$ is accepted as the new current feasible solution. If $S'$ is of higher cost, we reject this solution and check another randomly chosen neighbour of the current feasible solution. This allows us to randomly *walk* around $\Sigma$, without reducing the goodness of our current solution. Hill climbing has the potential to get stuck in a local minimum (or *freeze*), so stopping heuristics are required. To increase the chance of forming a good solution we repeat the random walk (or *trial*) a number of times, each time beginning with a random initial feasible solution.

In the hill climbing algorithm the current feasible solution is an approximation $S$ to a covering array in which certain $t$-subsets are not covered. The cost function is based on the number of $t$-subsets that are not covered, so that a covering array itself will have a cost of zero. A potential transformation is made by selecting one of the $k$ sets belonging to $S$ and then replacing a random point in this $k$-set by a random point not in the $k$-set. The number of blocks remains constant throughout the hill climbing trial.

We have used the method described by Stardom [22] to determine our array size. We set loose upper and lower bounds on the size of an optimal array and then use a binary search process to find the smallest sized covering array in this interval. An alternative method is to start with the size of a known test suite and search for a solution. This of course uses less computational resources, but the required test suite size must be known ahead of time. Ideally in a real system this is the method which we would use. For the results in this paper we have used the first method, since we do not yet know what size test suites are obtainable.

Clearly good data structures are required to enable the relative cost of the new feasible solution to be calculated efficiently, and the transition (if accepted) to be made quickly.

### 3.6. Simulated Annealing

Simulated annealing has been used by Nurmela and Östergård [18], to construct covering designs which have a structure very similar to covering arrays.

Simulated annealing uses the same approach as hill climbing but allows the algorithm, with a controlled probability, to make choices that reduce the quality of the current solution. The idea is to avoid getting stuck in a bad configuration while continuing to make progress. If the transformation results in a feasible solution $S'$ of higher cost, then $S'$ is accepted with probability $e^{-(c(S')-c(S))/K_B T}$, where $T$ is the controlling temperature of the simulation and $K_B$ is a constant. The temperature is lowered in small steps with the system being allowed to approach "equilibrium" at each temperature through a sequence of transitions (or Markov chain) at this temperature. Usually this is done by setting $T := \alpha T$, where $\alpha$ (the *control decrement*) is a real number slightly less than 1. After an appropriate stopping condition is met, the current feasible solution is taken as an approximation to the solution of the problem at hand. Again, we improve our chances of obtaining a good solution by running a number of trials.

Stardom [22] has recently compared simulated annealing with other types of local search, viz. tabu search and genetic algorithms, for finding covering arrays of strength 2. Stardom has reported several new upper bounds (including $CAN(2,16,8) \leq 145$ and $CAN(2,18,11) \leq 225$) using a simulated annealing algorithm. We have improved on some of those bounds using our simulated annealing algorithm. For instance we have found $CAN(2,16,6) \leq 62$. (For a table of known upper bounds for $t = 2$ see [22].)

### 3.7. Great Deluge Algorithm

One further heuristic search technique is the *great flood* or *great deluge* algorithm [11], and a variant thereof called *threshold accepting*. These follow a strategy similar to simulated annealing but often display more rapid convergence. Instead of using probability to decide on a move when the cost is higher, a worse feasible solution is chosen if the cost is less than the current threshold[1]. As the algorithm progresses, the threshold is reduced moving it closer to zero. Our current research focuses on the application of these algorithms to covering arrays.

### 3.8. Using an Integrated Approach

Finding the best covering array for a non-trivial problem is difficult. It is possible that a simulated annealing

[1]This threshold value is sometimes referred to as the *water level* which, in a profit maximizing problem, would be rising rather than falling (as is happening in this case).

algorithm, or a greedy method such as AETG, is the best approach, but further comparisons need to be explored.

AETG has been attributed with the best known upper bound of 180 for $CA(2, 20, 10)$. This array does not have a tight bound using an algebraic construction [22]. The AETG patent describes the use of some simple constructions, the merging of smaller test suites and a post processing stage to reduce test cases [6].

In general one would expect the algebraic constructions to use less computational power than the given algorithms. We have not extensively compared the performance of the individual algorithms in this paper, but note that the time to construct test suites is one factor that must be considered and compared when evaluating methods for building these objects. Table 6 gives some sample user CPU times for our versions of three algorithms. Stardom [22] has compared several methods of computational search, but his results are restricted to fixed level arrays and he did not include the TCG or AETG algorithms.

Given the computational efficiency of an algebraic construction, it is the best method to find a covering array when one is known to exist for the given parameters $k, t, v$. Indeed this idea has been suggested by Stevens et al. [23] who point out that this may not be a simple task. In order to use an algebraic construction we often use smaller objects which must also be constructed. The recursive nature of this makes the existence question alone here quite difficult (see [9]). If we can determine that the smaller objects exist, we first need to build these, and the information on how to do so must be stored somewhere. In addition, we may also need to store many small starter objects, for which space constraints may become a problem. A further issue is that there are many different types of constructions and sometimes multiple ways to arrive at the same object. This perhaps explains why commercial test generators do not yet utilize the best known constructions, but instead search each time from scratch.

By combining several of these techniques we expect to be able to find a large range of arrays that can be expanded or reduced as necessary. We could for instance, begin with a less costly algorithm, such as the TCG and then define a critical point in our test suite where we make a switch to a more computationally expensive algorithm. Another possibility is to simply build a starter test suite deterministically, and then use simulated annealing to reduce its size. An example of this technique is described in [6].

## 4. Constructing Variable Strength Covering Arrays

Variable strength covering arrays can be viewed as a collection of covering arrays inside of a larger covering array. We could begin by building each individual array separately and then combine these in order to gain the additional coverage needed for the whole system. Or we could begin with a covering array for the whole system and alter it to obtain the higher strength coverage required for the designated component subsets.

| Variable Strength Arrays | | |
|---|---|---|
| Base Array | Subset of Higher Coverage | Test Suite Size |
| $CA(N; 2, 11, 6)$ | $CA(N; 3, 6, 6)$ | 302 |
| $CA(N; 2, 20, 4)$ | $CA(N; 3, 9, 4)$ | 125 |
| $CA(N; 2, 10, 3)$ | $CA(N; 3, 6, 3)$ | 33 |
| $MCA(N; 2, 5^1 4^4 3^{11} 2^5)$ | $MCA(N; 3, 5^1 4^2 3^5)$ | 80 |
| $MCA(N, 2, 3^5 4^{10} 5^2)$ | $MCA(N; 3, 3^2 4^2 5^2)$ | 100 |

**Table 4. Test Suite Sizes for Variable Strength Arrays Using Simulated Annealing**

We outline an ad hoc construction here for the example given in Table 3, and investigate systematic methods for handling variable strength arrays in a subsequent paper. In order to construct the variable strength array we use our AETG algorithm to find a strength two covering array. We then try to remove duplicate pairs in a way that leads to new, uncovered triples. For instance, in the diagram in Table 3 we can see that had the last row of the 2-way coverage been (0,3,6,8) instead of (0,3,6,9), it could have been changed to (0,3,6,9) with the resulting system still providing pairs (0,8), (3,8), (6,8). Thus we could replace the 8 with a 9 without reducing our pair coverage, but covering a new triple. This step could be performed with a simulated annealing algorithm. Lastly we append the missing triples with a random choice of values for the other components.

Another way to approach this is to begin by enumerating all of the interactions for the higher strength subsets. For instance, we could build a mixed level covering array of strength three for components B, C and D and then fill horizontally by adding a column and the symbols needed so that all missing pairs are covered.

Lastly we might simply extend our simulated annealing algorithm to compute the cost as a function of both levels of interaction and build the suite directly in that manner. We are currently working on a program that uses annealing to build these arrays. Table 4 provides some preliminary test suite sizes for variable strength arrays with two different strengths.

The best method for building a variable strength test suite is still open and the discovery of good algorithms and constructions for these is an interesting problem.

## 5. Computational Results

We have implemented our own versions of the TCG algorithm for pair-wise coverage and the AETG algorithm for $t$-way coverage. In addition we have implemented both a

| | Minimum Test Cases in Test Suite | | | | | |
|---|---|---|---|---|---|---|
| | TCG[1] | AETG[1] | Our-TCG | Our-AETG | HC | SA |
| MCA-1 | 20 | 19 | 18 | 20 | 16 | 15 |
| MCA-2 | 45 | 45 | 42 | 44 | 42 | 42 |
| MCA-3 | 30 | 30 | 25 | 28 | 23 | 21 |
| MCA-4 | 33 | 34 | 32 | 35 | 30 | 30 |

**Table 5. Comparisons for 2-way Coverage**

1. Source = Yu-Wen et al.[30]
MCA-1. $MCA(N; 2, 5^1 3^8 2^2)$
MCA-2. $MCA(N; 2, 7^1 6^1 5^1 4^5 3^8 2^3)$
MCA-3. $MCA(N; 2, 5^1 4^4 3^{11} 2^5)$
MCA-4. $MCA(N; 2, 6^1 5^1 4^6 3^8 2^3)$

| | CPU User Time in Seconds | | |
|---|---|---|---|
| | Our-TCG | Our-AETG | SA |
| MCA-1 | 6 | 58 | 214 |
| MCA-2 | 57 | 489 | 874 |
| MCA-3 | 33 | 368 | 379 |
| MCA-4 | 42 | 376 | 579 |
| CA-1 | 1,333 | 6,001 | 10,833 |
| CA-2 | NA | 359 | 13,495 |

**Table 6. Comparisons of Runtimes**

MCA-1. $MCA(N; 2, 5^1 3^8 2^2)$
MCA-2. $MCA(N; 2, 7^1 6^1 5^1 4^5 3^8 2^3)$
MCA-3. $MCA(N; 2, 5^1 4^4 3^{11} 2^5)$
MCA-4. $MCA(N; 2, 6^1 5^1 4^6 3^8 2^3)$
CA-1. $CA(N; 2, 20, 10)$
CA-2. $CA(N; 3, 6, 6)$

hill climbing and simulated annealing program for handling $t$-way coverage. These algorithms use standard combinatorial techniques to store a $t$-set as a rank (an integer representation of a subset), which provide a generic way of representing different size sets. The simulated annealing program is based on that of [18]. While our techniques are fully general, we have emphasized the fixed level cases in our reporting so that we can make comparisons with results in the literature. Sample performance results for the TCG, AETG and simulated annealing algorithms are included to give a flavour for the time required to run each of these. All of the programs are written in C++ and run on Linux using an INTEL Pentium IV 1.8 GHZ processor with 512 MB of memory. Variation in the runtimes of these algorithms depends on parameter settings and number of iterations performed.

Although the implementation of our AETG algorithm is as described in the literature [5], it should be acknowledged that the actual commercial product is patented and may include some simple construction techniques as well as post-processing stages. These are not included in our implementation. One additional heuristic has been added in our version of the AETG algorithm, for the case of $t > 2$. In the algorithm described in [5], it is unclear what happens when

| | Minimum Test Cases in Test Suite | | | | | |
|---|---|---|---|---|---|---|
| | IPO[1] | AETG[1] | Our-TCG | Our-AETG | HC | SA |
| CA-1 | 9 | 9 | 9 | 9 | 9 | 9 |
| MCA-1 | 19 | 15 | 17 | 17 | 16 | 16 |
| MCA-2 | 36 | 41 | 34 | 37 | 30 | 30 |
| MCA-3 | 29 | 28 | 26 | 27 | 21 | 21 |
| CA-2 | 66 | NA | 56 | 56 | 47 | 45 |
| CA-3 | 218 | 180 | 213 | 198 | 189 | 183 |

**Table 7. Comparisons for 2-way Coverage**

1. Source = Yu et al.[29]
CA-1. $CA(N; 2, 4, 3)$
MCA-1. $MCA(N; 2, 3^{13})$
MCA-2. $MCA(N; 2, 4^{15} 3^{17} 2^{29})$
MCA-3. $MCA(N; 2, 4^1 3^{39} 2^{35})$
CA-2. $CA(N; 2, 100, 4)$
CA-3. $CA(N; 2, 20, 10)$

choosing a symbol between 2 and $t - 1$. The first symbol is always chosen as one of the symbols found in the most uncovered $t$-sets. We maintain this first step, then continue to choose from a symbol in the same set until we have the first $t - 1$ symbols filled in. We then continue to follow the algorithm as stated. Our results for $t = 3$ seem to improve on those reported in the literature for the commercial AETG program (see Table 9). Our AETG algorithm runs three hundred times reporting only the smallest array found at the end. All of these iterations were counted as part of the total time reported in Table 6.

For our version of the TCG algorithm, a few minor refinements were applied. The TCG algorithm as described in [30] does not fully describe how ties are handled. We chose to break these randomly in our version of the algorithm. In addition, TCG uses the *least used* symbols as a heuristic when choosing which symbol to include in the test suite. We tightened this definition to count a symbol as being used *only* in the case of it contributing to a new uncovered interaction pair. Our algorithm runs 5,000 times keeping only the best array at the end. All 5,000 iterations were included in the total time reported in Table 6.

The performance results presented for simulated annealing reflect the total time taken to find all arrays through a binary search process. Therefore the numbers reported in Table 6 may be reduced if tighter bounds are used as a starting point.

For all of the algorithms we ran a series of trials, but report only the best test suite obtained (the one with the smallest number of rows). We use the abbreviations *HC* for hill climbing and *SA* for simulated annealing in Tables 5-9. Table 5 compares our results with those reported in [30]. Our TCG and AETG algorithms produce similar results to those reported. In all of the four test suites, however both our hill climbing and simulated annealing algorithms improve significantly on the bounds given by these other algorithms.

| | Minimum Test Cases in Test Suite | | |
|---|---|---|---|
| | Stardom[1] | Our-AETG | SA |
| CA-1 | 65 | 70 | 62 |
| CA-2 | 88 | 94 | 87 |
| CA-3 | 113 | 120 | 112 |
| CA-4 | 116 | 123 | 114 |

**Table 8. Comparisons for 2-way Coverage**

1. Source = Stardom[22]

CA-1. $CA(N; 2, 16, 6)$
CA-2. $CA(N; 2, 16, 7)$
CA-3. $CA(N; 2, 16, 8)$
CA-4. $CA(N; 2, 17, 8)$

The hill climbing and simulated annealing algorithms both produced similar lower bounds, but in our experimentation we found that quite often the simulated annealing produced these in fewer trials. More experimentation is needed here.

Table 7 compares our results with those reported in [29]. For these examples our algorithms produce arrays at least as small as those produced by the IPO algorithm, although in two cases the reported commercial AETG results are smaller than ours.

Simulated annealing consistently does as well or better than hill climbing, so we report only those results for the next two tables. Table 8 compares results for some fixed level arrays reported in [22]. We have done as well or better than these. Since the results in [22] were obtained from a similar algorithm, we attribute this to the need for better tuning of the parameters of an annealing algorithm. We have not yet fine tuned the cooling schedule which plays a role in the quality of the final results. We plan to do this in the next phase of our investigation.

Lastly, Table 9 compares our results against some known strength-three algebraic constructions reported in [4]. In this case the expectation was that it would be difficult to match the known results in an initial implementation of these algorithms. Here the surprise was that our AETG algorithm found smaller arrays than those reported using the commercial AETG product. This may be due to the additional heuristic added when choosing the first $t-1$ symbols for each test suite. As expected, our simulated annealing algorithm did not perform as well as most of the algebraic constructions. In the case of a $CA(N; 3, 6, 6)$, however, we have found a smaller array using simulated annealing. Further experimentation is needed with a more refined algorithm. Of course, in many cases constructions are not known (or may not exist) such as is true in the last two entries of this table. For these arrays, simulated annealing finds an *optimal* solution. There are very few known constructions for mixed-level covering arrays. Therefore a fixed level array of a larger size would need to be constructed and used in its place. This may require more

| | Minimum Test Cases in Test Suite | | | |
|---|---|---|---|---|
| | Construction[1] | AETG[1] | Our-AETG | SA |
| CA-1 | 33 | 47 | 38 | 33 |
| CA-2 | 64 | 105 | 77 | 64 |
| CA-3 | 125 | NA | 194 | 152 |
| CA-4 | 305 | 343 | 330 | 300 |
| CA-5 | 1331 | 1508 | 1473 | 1426 |
| CA-6 | 185 | 229 | 218 | 201 |
| MCA-1 | NA | NA | 114 | 100 |
| MCA-2 | NA | NA | 377 | 360 |

**Table 9. Comparisons for 3-way Coverage**

1. Source = Chateauneuf et al.[4]
CA-1. $CA(N; 3, 6, 3)$
CA-2. $CA(N; 3, 6, 4)$
CA-3. $CA(N; 3, 6, 5)$
CA-4. $CA(N; 3, 6, 6)$
CA-5. $CA(N; 3, 6, 10)$
CA-6. $CA(N; 3, 7, 5)$
MCA-1.$MCA(N; 3, 3^2 4^2 5^2)$
MCA-2.$MCA(N; 3, 10^1 6^2 4^3 3^1)$

test cases than a mixed level array found by computational search. Additionally, a real test suite may include special test cases for particular configurations. Once again current constructions do not handle this issue. For this reason, it is interesting to return to the need for an integrated approach, perhaps one that combines algebraic constructions with more sophisticated search methods.

## 6. Conclusions

We have discussed some of the uses of combinatorial objects in testing interactions among software components, have raised several questions about the types of combinatorial objects that may be useful for the software tester and have suggested some ways to build these objects using known constructions and search techniques.

Preliminary results on hill climbing and simulated annealing for mixed level covering arrays are presented, although more experimentation and tuning of these algorithms is required. In addition, performance measures need to be applied to determine the overall usefulness of each approach.

We have presented a variable strength covering array which handles multiple levels of interactions. These arrays allow us to assign interaction weights to subsets of components. We have provided an ad hoc method to build these but have left the development of formal methods for future work. It is not entirely clear that even variable strength covering arrays will provide the flexibility that is really needed for testing combinatorial systems. We may want to allow some components to be involved in more than one grouping, or perhaps to test individual groupings alone and then

to combine these while retaining the original tests.

Lastly, we may think about how this model can be extended to a broader definition of component testing. In this paper we have restricted ourselves to the idea of finding faults when components are tested in combination. We may find we can use this model for quantitative testing, such as determining which combinations cause performance barriers.

## Acknowledgments

Research is supported by the Consortium for Embedded and Internetworking Technologies and by ARO grant DAAD 19-1-01-0406.

## References

[1] I. Anderson. *Combinatorial Designs and Tournaments*. Oxford University Press, New York, 1997.

[2] R. Brownlie, J. Prowse, and M. S. Padke. Robust testing of AT&T PMX/StarMAIL using OATS. *AT&T Technical Journal*, 71(3):41–7, 1992.

[3] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proc. of the Intl. Conf. on Software Testing Analysis & Review*, 1998, San Diego.

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

[5] 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.

[6] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. Method and system for automatically generating efficient test cases for systems having interacting elements *United States Patent*, Number 5,542,043, 1996.

[7] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–8, 1996.

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

[9] 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.

[10] 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.

[11] G. Dueck. New optimization heuristics - the great deluge algorithm and the record-to-record travel. *Journal of Computational Physics*, 104:86–92, 1993.

[12] 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.

[13] L. Gargano, J. Körner, and U. Vaccaro. Capacities: from information theory to extremal set theory. *J. Combin. Theory Ser. A*, 68(2):296–316, 1994.

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

[15] G. Katona. Two applications (for search theory and truth functions) of Sperner type theorems. *Periodica Math.*, 3:19–26, 1973.

[16] D. Kleitman and J. Spencer. Families of k-independent sets. *Discrete Math*, 6:255–262, 1973.

[17] R. Mandl. Orthogonal latin squares: an application of experiment design to compiler testing. *Communications of the ACM*, 28(10):1054–8, 1985.

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

[19] P. R. J. Östergård. Constructions of mixed covering codes. Technical report, Digital Systems Laboratory, Helsinki Univ. of Technology, 1991.

[20] A. Réyni. *Foundations of Probability*. Wiley, New York, 1971.

[21] N. Sloane. Covering arrays and intersecting codes. *Journal of Combinatorial Designs*, 1(1):51–63, 1993.

[22] J. Stardom. Metaheuristics and the search for covering and packing arrays. Master's thesis, Simon Fraser University, 2001.

[23] B. Stevens and E. Mendelsohn. Efficient software testing protocols. In *Proc. of Center for Advanced Studies Conf. (Cascon '98)*, 1998, pp. 270-293, Ontario.

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

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

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

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

[28] 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.

[29] L. Yu and K. C. Tai. In-parameter-order: a test generation strategy for pairwise testing. In *Proc. Third IEEE Intl. High-Assurance Systems Engineering Symp.*, 1998, pp. 254-261.

[30] 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.