A Deterministic Density Algorithm for Pairwise Interaction Coverage

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

Myra B. Cohen
Computer Science
University of Auckland
Private Bag 92019
Auckland, New Zealand
myra@cs.auckland.ac.nz

Renée C. Turban
Computer Science
Arizona State University
P.O. Box 875406
Tempe, Arizona 85287
renee.turban@asu.edu

ABSTRACT
Pairwise coverage of factors affecting software has been proposed to screen for potential errors. Techniques to generate test suites for pairwise coverage are evaluated according to many criteria. A small number of tests is a main criterion, as this dictates the time for test execution. Randomness has been exploited to search for small test suites, but variation occurs in the test suite produced. A worst-case guarantee on test suite size is desired; repeatable generation is often necessary. The time to construct the test suite is also important. Finally, testers must be able to include certain tests, and to exclude others.

The main approaches to generating test suites for pairwise coverage are examined; these are exemplified by AETG, IPO, TCG, TConfig, simulated annealing, and combinatorial design techniques. A greedy variant of AETG and TCG is developed. It is deterministic, guaranteeing reproducibility. It generates only one candidate test at a time, providing faster test suite development. It is shown to provide a logarithmic worst-case guarantee on the test suite size. It permits users to "seed" the test suite with specified tests. Finally, comparisons with other greedy approaches demonstrate that it often yields the smallest test suite.

KEY WORDS
software interaction testing, covering array, greedy algorithm, pairwise coverage

1 Introduction

Software systems are built using components. Consequently, system faults can result from unexpected interactions among components. In an Internet-based software system, for example, customers may use a variety of browsers, operating systems, connection types, and printer configurations. An example is shown in Table 1. In order to test this system exhaustively, we must test the software on all of the possible supported configurations. To test all possible interactions for the system in Table 1, we would need $3^4 = 81$ configurations.

This may be feasible for such a small system but the number of necessary tests suffers a combinatorial explosion. With ten components each having four possible settings, 4^{10}=1,048,576 test configurations are needed! Each component is a *factor* affecting the system, and each setting of the component is a *value* or *level* for the factor. In view of the infeasibility of exhaustive testing, we can instead ask for a guarantee that we test all pairs of interactions or all *n*-way interactions [1, 2, 3, 4]. In the example shown in Table 1 we can cover all pairs of interactions using only nine different configurations (see Table 2). With ten factors each with 4 possible values we can cover all pairs of interactions using at most 25 configurations.

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

Table 1. Four Factors, Each With Three Values

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 [5]. In further work, Burr *et al.* provide more empirical results to show that this type of test coverage is effective [6]. Dunietz *et al*. link the effectiveness of these methods to software code coverage. They show that high code block coverage is obtained when testing all two-way interactions, but higher subset sizes are needed for good path coverage [2]. Kuhn *et al*. examined fault reports for three software systems. They show that 70% of faults can be discovered by testing all two-way interactions, while 90% can be detected by testing all three way interactions. Six-way coverage was required in these systems to detect 100% of the faults reported [7].

In this paper, we first introduce some combinatorial definitions to provide a vernacular for examining test suite generation. We then examine in more detail criteria to evaluate methods for this generation, and discuss the importance of each. Existing techniques are evaluated according to these criteria, and we find that while each of the available techniques exhibits some desirable properties, none meets

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	Mozilla	Windows	PPP	Networked
8	Mozilla	Linux	LAN	Screen
9	Mozilla	Macintosh	ISDN	Local

Table 2. Test Suite Covering All Pairs from Table 1

all of the criteria identified. We then suggest a strategy for satisfying the criteria. A conceptually simple algorithm results that exhibits nice theoretical properties, especially that it ensures a logarithmic worst-case guarantee on test suite size. In order to assess whether the method is also competitive in a practical sense, we describe a careful implementation of it, and examine a wide variety of sample construction problems, concluding that the method proposed is faster and often more accurate than existing techniques.

2 Combinatorial Models

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* λ times [8]. 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 [3] 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.

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* λ 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, $CAN(2, 5, 3) = 11$ [9, 10].

We can map a covering array to a software test suite as follows. In a software test we have k *components* or *factors*. 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 we have $t = 2$,$k = 4$, $v = 3$, and $N = 9$. Each component is represented by one column; each row is an individual test of the test suite. All pairs of components between any two columns are tested in this test suite.

Covering arrays only suit the needs of software testers when all factors have the same number of values. However, this is often not the case. For instance one factor can have four possible values and one only two. Indeed, this is a normal occurrence.

The variation among factor levels can be handled with the *mixed level covering array*. Several authors have suggested its use for software testing (see [9, 11, 12]), but few results are known about upper bounds and how to construct these.

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 $i \quad (1 \leq i \leq k)$ 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.

We use a shorthand notation to describe mixed level covering arrays by combining equal entries in $(v_i : 1 \leq i \leq k)$. For example three entries each equal to 2 can be written as 2^3. We can write an $MCA(N; t, k, (v_1 v_2 ... v_k))$ as an $MCA(N; t, (w_1^{r_1} w_2^{r_2} ... w_s^{r_s}))$ where $k = \sum_{i=1}^{s} r_i$ and $(w_j : 1 \leq j \leq k) \subseteq \{v_1, v_2, ..., v_k\}$.

3 Comparing Methods for Test Suite Generation

The variety of methods for generating test suites for pairwise coverage arises from a number of different objectives to be addressed. We identify a number of criteria that have been previously considered. Evidently, *completeness of coverage* is a primary goal; we interpret this to be a requirement for the test suite to be a covering array of strength two. In order to accelerate execution of the test suite, *small test suites* are desired. Naturally both good performance in the average case *and* a *worst-carse guarantee* on test suite sizes are desirable. At the same time, *efficient construction* of the test suite is needed to reduce time spent constructing, rather than executing, the test suite. Different testers can have quite different experiences with the same method, and suffer the difficulty that a test suite size is reported once that the method does not easily reproduce at a later time. In many applications, however, *predictability* of the test suite's size and structure is desirable.

It is rarely the case even with automatic test generation that a tester does not wish to impose certain additional constraints. For example, the ability to *seed* a test suite by specifying the inclusion of certain tests is often necessary. In addition, constraints among the factors can dictate that the test suite *avoid* certain tests; these 'avoids' can be in

the form of pairs that need not be covered, or of pairs that cannot be covered.

These many criteria have led to a wide variety of approaches, which we review briefly next. Many algebraic and combinatorial constructions for orthogonal arrays and covering arrays appear in the literature (see [8, 9, 10] for example). Combinatorial constructions are only known for some parameter sets; this severely limits the applicability to practical testing problems. Nevertheless, Williams *et al.* [3, 12] develop a strategy, *TConfig*, for employing a recursive construction based on orthogonal arrays to construct test suites. Their method is both fast and general. In addition, it provides a worst-case guarantee on test suite size that is optimal up to a constant factor. It does not provide for seeds or avoids. While it shares the speed of combinatorial constructions and permits more general application, it generates test suites that are often much larger than needed. This problem is especially prevalent in mixed level covering arrays, since the method is designed for fixed level problems.

To address concerns with test suite size, many computational methods have been explored. Exhaustive generation is intractable, and optimization methods such as linear programming have proved successful only on small problems [10]. Computational search techniques to find covering arrays include techniques such as hill climbing and simulated annealing; other sophisticated search techniques have proved less successful until this time [11]. Both hill climbing and simulated annealing provide general methods that appear to produce the smallest test suites across a wide range of problems [13]. They provide for seeds. However, predictability is a major concern, and the time to construct a test suite can be prohibitive. They also fail to produce a worst-case guarantee on test suite size. Nevertheless, if minimizing the size of the test suite is of paramount concern, simulated annealing currently appears to be the best method available.

Most of the available techniques sacrifice some effort in minimizing the test suite in order to get a simpler technique. Hence the majority of current techniques for mixed level arrays use greedy methods to find test suites [1, 4, 14].

The AETG system and the Test Case Generator (TCG) [1, 15, 14] 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 [1, 15, 14]). 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. Tung *et al.* [14] suggest a deterministic algorithm. The authors begin with a deterministic ordering of the factors. Another greedy algorithm, *In-Parameter-Order (IPO)*, has the benefit of reusing old test cases when new factors are added. It does this by expanding in a vertical and horizontal fashion [4].

4 The Logarithmic Guarantee

Let us explore these in more detail. Suppose that we are to test a system with k factors $f_1, \ldots, f_k$. The factor f_i is permitted to take on any of v_i *levels* or *values*, which we denote by $\{\sigma_{i,j} : j = 1, \ldots, v_i\}$. The objective is to produce an $MCA(N; 2, k, (v_1 ... v_k))$.

The AETG system attempts to make a 'small' covering array using a greedy strategy. It selects a single test at a time, repeating this until all pairs are covered in at least one of the selected tests. Since the objective is to minimize the number of tests, AETG concentrates on the selection of each test to maximize the number of previously uncovered pairs that are covered by this test. The paper makes two main contributions [1]:

1. It shows a *logarithmic* bound on the number of tests needed as a function of k.

2. It describes a (greedy) heuristic for the selection of tests.

The first relies on a conceptually simple construction method for covering arrays. Having selected some (partial) collection of tests, we record the pairs $\mathcal{P}$ yet to be covered. Among the $\prod_{i=1}^{k} v_i$ possible tests, there can be substantial variation in the number of pairs in $\mathcal{P}$ that the test covers. We select a test that covers that largest number of pairs in $\mathcal{P}$, add it to the collection of tests, and repeat this step until $\mathcal{P} = \emptyset$; at this point, we have the covering array. This is a greedy method, and by no means guarantees the minimum size of the possible test suite constructed. However, it does ensure that at each stage, at least $|\mathcal{P}|/L$ new pairs are covered where L is the product of the two largest of the sizes $\{v_i\}$. This in turn ensures that the size of the test suite constructed is bounded by a logarithmic function of the number k of factors (see [1] for details).

However, the authors do not propose an algorithm for finding the test that covers a maximum number of uncovered pairs, and instead adopt a greedy heuristic to produce each new test in turn. We review their method here. Each test is selected from a pool of M candidate tests, where M is a constant chosen in advance. To generate each candidate, first select a factor f_i and a value σ_i for this factor so that the choice of σ_i for f_i appears in the maximum number of uncovered pairs. Set $\pi(1) = i$. Then choose a random permutation of the indices of the remaining factors to form a permutation $\pi : \{1, \ldots, k\} \to \{1, \ldots, k\}$. Now assume that values $\tau_1, \ldots, \tau_i$ have been selected for factors $f_{\pi(1)}, \ldots, f_{\pi(i)}$. Select a value τ_{i+1} for factor $f_{\pi(i+1)}$ by selecting that value which yields the maximum number of new pairs with the selected values for the i factors already fixed.

Repeating this process M times exploits the randomness in the factor ordering, and yields different tests from

which to select. Naturally, one selects the best in terms of newly covered pairs, and adds it to the test suite.

While the authors note that this appears to exhibit a logarithmic performance in practice, their earlier guarantee does not apply because the test selection does not ensure that a selected test covers the maximum possible number of new pairs. In view of the next result, this is not surprising.

Given a collection $\mathcal{P}$ of uncovered pairs, and a specified number p, it is NP-complete to determine if there exists a test covering at least p pairs. See [16]. Membership in NP is straightforward, since one can nondeterministically select a test, and compute in polynomial time (deterministically) the number of pairs of $\mathcal{P}$ covered. To establish NP-hardness, we give a reduction from MAX 2SAT ("Given a logical formula in 2-conjunctive normal form, and an integer p, is there a truth assignment to the variables that makes at least p clauses true?"). This problem is NP-complete (see, for example, [17]). Let $\mathcal{F}$ be a formula in 2-conjunctive normal form with k logical variables. We form k factors, each with two levels, 'true' and 'false'. For each clause of $\mathcal{F}$, treat as a covered pair the truth assignment to the two variables which makes the clause false. Then $\mathcal{P}$ contains all pairs not covered in this way. Now we determine whether there is a test which covers at least p pairs of $\mathcal{P}$. A test corresponds directly to a truth assignment for $\mathcal{F}$, and an uncovered pair to a clause which evaluates to true. Thus the existence of such a test is equivalent to a solution to MAX 2SAT.

In plain terms, what this says is that an efficient technique to select a test covering the maximum number of uncovered pairs is unlikely to exist. However, this leaves an unsatisfactory situation. The current proof of logarithmic growth hinges on selecting such a test! Fortunately, the situation is not as bad as it first appears. Indeed a more careful reading of the proof of logarithmic growth establishes that one does not need to find a test which covers the maximum number of uncovered pairs. All one needs is to find a test that covers the *average* number of uncovered pairs.

It appears that the test selection in AETG does not ensure this, although for practical purposes unless M is quite small, the likelihood that at least one of the M candidates has this property is high. Nevertheless, it is reasonable to ask for a test selection technique that *guarantees* to cover at least the average number. We pursue this next.

The lack of a guarantee results primarily from the greedy nature of the test selection. In particular, when selecting a value for the ith factor, only its interaction with the first $i-1$ factors is considered. This can (and does) result in a selection which make selections for the later factors less desirable. Indeed if there are 100 factors, and we are selecting a value for the fifth, for example, its interaction with the later 95 factors is arguably more important than its interaction with the first four. We use this intuition to suggest an alternate approach.

We consider the construction of a test suite with k factors. The number of levels for factor i is denoted by v_i. For factors i and j, we define the *local density* to be $\delta_{i,j} = \frac{r_{i,j}}{v_i v_j}$ where $r_{i,j}$ is the number of uncovered pairs involving a value of factor i and a value of factor j. In essence, $\delta_{i,j}$ indicates the fraction of pairs of assignments to these factors which remain to be tested. We define the *global density* to be $\delta = \sum_{1 \le i < j \le k} \delta_{i,j}$. At each stage, we endeavour to find a test covering at least δ uncovered pairs.

To select such a test, we repeatedly fix a value for each factor, and update the local and global density values. At each stage, some factors are *fixed* to a specific value, while others remain *free* to take on any of the possible values. When all factors are fixed, we have succeeded in choosing the test. Otherwise, select a free factor f_s. We have $\delta = \sum_{1 \le i < j \le k} \delta_{i,j}$, which we separate into two terms:

$$\delta = \sum_{\substack{1 \le i < j \le k \\ i,j \ne s}} \delta_{i,j} + \sum_{\substack{1 \le i \le k \\ i \ne s}} \delta_{i,s}.$$

Whatever level is selected for factor f_s, the first summation is not affected, so we focus on the second.

Write $\rho_{i,s,\sigma}$ for $\frac{1}{v_i}$ times the number of uncovered pairs involving some level of factor f_i, and level σ of factor f_s. Then rewrite the second summation as

$$\sum_{\substack{1 \le i \le k \\ i \ne s}} \delta_{i,s} = \frac{1}{v_s} \sum_{\sigma=1}^{v_s} \sum_{\substack{1 \le i \le k \\ i \ne s}} \rho_{i,s,\sigma}.$$

We choose σ to maximize $\sum_{\substack{1 \le i \le k \\ i \ne s}} \rho_{i,s,\sigma}$. It follows that $\sum_{\substack{1 \le i \le k \\ i \ne s}} \rho_{i,s,\sigma} \ge \sum_{\substack{1 \le i \le k \\ i \ne s}} \delta_{i,s}$. We then fix factor f_s to have value σ, set $v_s = 1$, and update the local densities setting $\delta_{i,s}$ to be $\rho_{i,s,\sigma}$. In the process, the density has not been decreased (despite some possible – indeed necessary – decreases in some local densities).

We iterate this process until every factor is fixed. The factors could be fixed in *any order at all*, and the final test has density at least δ. Of course it is possible to be greedy in the order in which factors are fixed. This has some practical value as we see later, but does not affect the logarithmic growth.

If we apply this method to the case where each factor has the same number of levels, the density is the average number of uncovered pairs in a test that could be selected, and we guarantee to select a test with at least this number of uncovered pairs.

5 A Deterministic Density Algorithm

While the density argument developed establishes that greedy methods can indeed yield a worst-case guarantee that is logarithmic, it does not address the question of whether controlling the worst case has a negative impact on the expected size of test suites. In this section we describe a practical implementation of the deterministic density algorithm (DDA).

AETG [1] overcomes the lack of lookahead by randomly reordering the factors, and selecting the best new test

from a set of candidates. While this randomness makes predictability problematic, it does in practice deal with problems resulting from factor ordering in a greedy method. Nevertheless, as argued in [14], predictability is important. With AETG it is also the case that the accuracy obtained by considering more candidates comes at the price of greater computation time. TCG [14] addresses the concern with predictability by making deterministic selections. It always fixes factors in nonincreasing order by number of levels for the factor. Within this factor order, it attempts to make the best selection of value for each factor in turn (measured as number of pairs covered that were previously uncovered). In order to avoid the myopia of considering only one candidate, however, it retains a set of best candidates so far. The number of such candidates equals the largest number of values for a factor. As each is extended, only the best are retained; once candidate tests are completed, the best is selected for inclusion in the test suite. The determinacy of the approach addresses not only the issue of predictability, but to a certain degree also lessens computational resources. Test results in [13, 14] indicate that TCG sometimes maintains the accuracy of AETG, but it can perform poorly with respect to the accuracy of AETG. In part, the fixed ordering of factors in TCG removes a degree of freedom that AETG exploits to an extent.

In assessing the practical value of a deterministic density approach, certain decisions must be made along the lines indicated by AETG and TCG. Evidently it would be possible to use the density selection criterion in a randomized approach such as AETG, substituting the selection based solely on pairs covered with factors already fixed by the one based on density. We do not pursue this avenue for two reasons. Primarily, a randomized approach using density still faces problems with predictability. Moreover, density provides guidance as to the order in which factors should be fixed in order to cover the most (or the most important) pairs.

It would also be possible to adopt TCG's strategy of building 'in parallel' many candidates for the next test to be added to the suite. The overhead here is in computation time, and we expect that the time/accuracy tradeoff merits further exploration. As a proof of concept, however, we opt to implement a method that deterministically generates a single candidate test.

Staying within the logarithmic guarantee is ensured by selecting a next test that covers at least the average number of uncovered pairs. However, our intuition is that better results will be obtained by covering the largest number. While we have seen that this is NP-hard, it nonetheless indicates that in making choices in the algorithm, we ought to make choices to improve the density of coverage as much as is possible. First we consider the selection of factors. The density of a factor f_i can be taken as $\delta_i = \sum_{1 \le j \le k,\ j \ne i} \delta_{i,j}$. At every stage, we select the factor with the largest density to fix next. Once a factor is chosen, we need to select a value for this factor. To do this, for each possible value of the factor, we calculate the resulting density for the factor if the factor is fixed to the specified value. The value yielding the largest increase in density is selected, and the densities updated. In essence, we are simply using density as a surrogate for the number of pairs that become covered; in this way, we avoid the issue of not knowing which specific pairs are covered until after the test is selected.

0	4	6	$\delta_{2,0} = 1 + \frac{1}{4} = 1.25$	$MCA(N; 4^1 2^1 3^1)$
1		?	$\delta_{2,1} = 1 + \frac{2}{4} = 1.5$	$v_0 = \{0, 1, 2, 3\}$
.	.	.	$\delta_{2,2} = 1 + \frac{2}{4} = 1.5$	$v_1 = \{4, 5\}$
				$v_2 = \{6, 7, 8\}$
			Symbol chosen is 7	.

Figure 1. Selecting a Symbol

Making the 'best' selection locally ensures in particular that we always retain a set of choices in which we do at least as well as the average. However, a few moments' thought reveals some concerns, particularly for mixed level arrays. The measure of local density scales the number of uncovered pairs by the initial number to be covered. Hence between two factors with two levels each, every pair contributes $\frac{1}{4}$ to the global density, while if the two factors have ten levels each, a pair contributes only $\frac{1}{100}$. This runs counter to our expectation that the pairs in the first situation are much more easily covered than those in the second. To address this, we redefine local density in a way that does not affect the logarithmic guarantee. Let $v_{\max}$ be the largest number of levels for any factor. When we handle two factors each having more than one level, in the local density we replace the denominator by $v_{\max}^2$, making every pair equally important from a density viewpoint. When one factor has had its value chosen (i.e. has only one level now), and the other remains to be chosen, we employ the denominator $v_{\max}$. When both factors have been fixed, we employ the denominator 1. This corrects the method to focus on factors with many levels rather than factors with few. In Figure 1 we show one step in this algorithm. In this example we have already chosen the first test case and fixed the first factor of the second. The next factor to fix is the third one. We calculate the local densities for the three symbols of this factor and select the one with the largest local density. Since more than one has the largest local density we select the first.

We implemented the algorithm using local densities defined in this way. Although the results are competitive with AETG and TCG, using densities in this way does not exhibit a preference among pairs to be covered. Consider two factors. If we consider a possible value for the first, it may have many or few uncovered pairs with the second factor; local density as defined does not provide a greater reward for covering those pairs involving a value appearing in many uncovered pairs. Hence as a practical matter, we

	Minimum Size of a Test Suite				
Parameters	DDA	AETG	TCG	IPO	TConfig
$5^1 3^8 2^2$	21	19	20		
$7^1 6^1 5^1 4^5 3^8 2^3$	43	45	45		
$5^1 4^4 3^{11} 2^5$	27	30	30		
$6^1 5^1 4^6 3^8 2^3$	34	34	33		
$4^{15} 3^{17} 2^{29}$	35	41		36	40
$4^1 3^{39} 2^{35}$	27	28		29	30
3^{13}	18	15		19	
2^{100}	15	10		15	14
4^{40}	43			49	40
4^{100}	51			52	43
10^{20}	201	180		212	231

Table 3. Comparison with Published Results

refined the notion of local density further.

Our objective is to revise the definition of density so that between one factor and another, each pair as it becomes covered makes a smaller reduction in the density. In this way, selection of the best improvement in density leads to a preference to cover pairs involving values having many uncovered pairs remaining. Implementing this is straightforward. Choose an *inflation factor* α. For two factors f_i and f_j, and a value σ of f_i, we calculate the ratio of uncovered pairs involving σ and a value in f_j to the value $v_{\max}$ (or 1 if f_j has already been fixed). Summing over all values of σ and dividing by $f_{\max}$ (or 1 if factor f_i has been fixed), and then raising the result to the power α, gives the local density $\delta_{i,j}$.

It is routine to see that this makes the contribution of a pair to the density larger when it involves factors between which which many uncovered pairs remain. We do not describe a complete set of experiments here, but found that choosing inflation factor $\alpha = 2$ (i.e. squaring the previous contributions) gave a useful improvement. It is necessary to ensure that such a change does not negate the logarithmic guarantee, but it is easily seen that it only changes the guarantee by a constant factor.

6 Computational Results

In order to assess the practicality of this deterministic density approach, we implemented a simple C program, DDA, to compare the sizes of test suites obtained aganst those in the literature for TCG [14], AETG [1, 18, 15], IPO [19, 4], and TConfig [20, 3]. We have implemented the AETG and TCG approaches independently [13], but in order to provide a preliminary assessment of the practicality of the approach, we restrict ourselves to comparisons with the test suite sizes published by others. A more complete comparison with implementations of these alternative approaches is justified if DDA indeed proves competitive against published results.

Table 3 presents sizes of test suites for a collection of mixed-level covering arrays, and for a few fixed-level covering arrays, for which results appear in the literature. We shall not attempt to draw too many conclusions from such a small sample of cases, when data is available for only some methods in each case. However, a few simple observations are in order. TConfig, as expected, appears to be quite effective in fixed-level cases when the number of values is a prime or prime power, but does not fare as well when we depart from cases of this type. AETG constructs $M = 50$ candidates for each test and selects the best; the method by which the candidates is chosen involves randomly permuting the factors. TCG deterministically builds a number of candidates equal to the maximum number of levels for a factor, and chooses the best. Nevertheless, while DDA constructs only a single candidate for each test, the results shown suggest that it is competitive; we are currently exploring the extension of DDA to maintain multiple candidates and select the best. The time/accuracy tradeoff here is unclear, but may explain to a degree cases in which DDA is less competitive.

It would be incorrect to think that minimum size of the test suite is the only parameter of importance. Indeed, if this were the case, one could in theory employ an exhaustive search method rather than a greedy one. More realistically, one could employ hill-climbing or simulated annealing [13]. For example, for the mixed-level covering array $5^1 4^4 3^{11} 2^5$, simulated annealing yields a solution with only 21 test cases (compare to 27 for DDA, 30 for TCG and AETG, in Table 3). Nevertheless, this simulated annealing result took 579 seconds of compute time [13], while (on a different machine) DDA took .16 seconds. The point here is not to compare these running times directly, as different platforms and implementations are involved. Rather the point is to emphasize that greedy methods are intended for speed. With this in mind, we report some execution times on a SunBlade 1000 system for DDA. The longest running time in our test cases was 24.9 seconds for the $CA(2, 100, 4)$. The case 2^{100} took 7.41 seconds, the case $4^1 3^{29} 2^{35}$ took 5.81 seconds, the case $4^{15} 3^{17} 2^{29}$ took 4.12 seconds. the case 10^{20} took 1.1 seconds, and all others took less than 0.2 seconds each.

These results demonstrate that DDA can produce test suites of reasonable size in a modest amount of time. Comparisons with other methods are speculative at best. However, on the basis of published timings, it appears that TConfig is faster than DDA, TCG and IPO are in the same range of timing, and AETG is somewhat slower. We conclude only that DDA appears to be competitive in terms of execution time as well.

One further experiment was conducted. Deterministic density causes one to focus on those factors for which the most remains to be covered, and for these factors chooses the value for which the most remains to be covered. As such the method lends itself naturally to applications in which a portion of the covering array has been determined. For example, we used DDA to find a CA(28;2,15,4). We then seeded an $MCA(N; 2, 4^{15} 3^{17} 2^{29})$ with these *partial* tests, and allowed DDA to complete each partial test to a complete one, and then add additional tests to complete the

mixed covering array. The result had only 32 tests, an improvement upon the results in Table 3. This method may provide a vehicle for obtaining better results using deterministic density. Moreover, it suggests that the approach can be quite useful in seeding portions of a covering array to be completed by DDA.

Our objective here was to assess whether DDA yields not just an approach of theoretical interest but also a practical and competitive technique. On the basis of the accuracy and time reported here, we believe that the method shows definite promise.

7 Conclusions

Cohen *et al.* [1] establish that a greedy method that always selects the next test to maximize newly covered pairs gives a test suite whose size grow logarithmically with the number of factors. They design AETG as an heuristic method to attempt to select a next test that covers close to this maximum number, but there is no theoretical guarantee that their method indeed achieves this objective (although the practical results suggest that it typically does). TCG employs a deterministic strategy that often, but not always, outperforms AETG for accuracy but also provides no such guarantee. Indeed, of the methods discussed, only the recursive combinatorial method TConfig provides such a guarantee; however, TConfig appears to perform well in practice when the covering array parameters are of a restricted type.

These observations motivate two questions. The first is to select a test that covers the maximum number of uncovered pairs, and we have shown in a strong sense that an efficient algorithm to do this is unlikely to exist; the problem is NP-hard. The second question is to determine what is needed to get a logarithmic bound on the number of tests in a test suite. Employing the method of Cohen *et al.* [1], we observed that it suffices to cover the *average* number of uncovered pairs. A density calculation is developed, and used to ensure that at every stage, selections of factors and levels is made that ensures that the average number of pairs covered among the candidate tests that remain at each stage never declines. Consequently, when all factors have been assigned levels, the test covers at least the average number of new pairs. This guarantees the logarithmic bound.

While the logarithmic bound is of theoretical interest, and answers a question implied in [1], the practical value of the method that results is not addressed by this theoretical bound. We therefore developed a deterministic density algorithm (DDA) for generating test suites; instead of choosing factors and values so as not to decrease the density, we instead choose a "steepest ascent" technique, at each stage making selections to increase the density as much as possible. We have compared DDA with four published methods, and found it competitive with respect to size of test suites, and apparently competitive in terms of execution time.

Further comparisons are needed to draw definitive conclusions, particularly with respect to execution times. Moreover, the theory underlying DDA indicates that a wide variety of candidate selections can be made without losing the logarithmic bound. It is therefore possible to select one of the available choices that does not decrease density randomly, or to maintain a set of candidate partial tests for exploration. AETG uses both of these techniques [1], and TCG uses the latter [14]. We expect that a randomized approach based on densities would provide an improvement in accuracy, at the expense of losing the reproducibility of the current method. The value of maintaining multiple candidates selected deterministically is less clear, and is the subject of current research.

Acknowledgments

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

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] D. Kuhn and M. Reilly. An investigation of the applicability of design of experiments to software testing. *Proc. 27th Annual NASA Goddard/IEEE Software Engineering Workshop*, 2002, pp. 91-95.

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

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

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

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

[12] 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.

[13] 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)* , 2003, pp. 38-48, Portland.

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

[15] 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.

[16] G. Seroussi and N. Bshouti, Vector sets for exhaustive testing of logic circuits, *IEEE Transactions on Information Theory* 34 (1988) 513-522.

[17] M. X. Goemans and D. P. Williamson, Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming, *Journal of the ACM (JACM)*, 42(6):1115-1145, 1995.

[18] 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.

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

[20] A. W. Williams. Determination of test configurations for pair-wise interaction coverage. *Testing of Communicating Systems: Tools and Techniques, IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems (TestCom 2000)*, pp. 59-74.