A Framework of Greedy Methods for Constructing Interaction Test Suites

Renée C. Bryce and Charles J. Colbourn
Computer Science and Engineering
Arizona State University
Tempe, Arizona 85287-8809
{rcbryce,colbourn}asu.edu

Myra B. Cohen
Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
myra@cse.unl.edu

ABSTRACT

Greedy algorithms for the construction of software interaction test suites are studied. A framework is developed to evaluate a large class of greedy methods that build suites one test at a time. Within this framework are many instantiations of greedy methods generalizing those in the literature. Greedy algorithms are popular when the time for test suite construction is of paramount concern. We focus on the size of the test suite produced by each instantiation. Experiments are analyzed using statistical techniques to determine the importance of the implementation decisions within the framework. This framework provides a platform for optimizing the accuracy and speed of "one-test-at-a-time" greedy methods.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Algorithms, Measurement, Experimentation

Keywords

Covering arrays, greedy algorithm, mixed-level covering arrays, pair-wise interaction coverage, software interaction testing

1. INTRODUCTION

Software systems are complex, and exhaustive combinatorial testing involves an exponential number of possible parameter settings. Interaction testing is a method to reduce the number of tests performed, and can yield high coverage with few tests, thereby reducing testing cost [1, 15, 22, 23, 33]. We illustrate this next.

Consider the modular system in Table 1. There are three pieces of hardware, three operating systems, three network connections, and three memory configurations that can be integrated. Different end users may use different combinations of components. To exhaustively test all combinations requires 3^4= 81 test configurations. The four components are *factors*, and the three values for each factor are its *levels*.

We reduce the 81 tests required for exhaustive testing by employing pair-wise interaction testing. Instead of testing every combination, all individual pairs of interactions are tested. The resulting test suite is shown in Table 2, and has only 9 tests. Each possible pair of values for factors appears in at least one test.

Table 1: A small modular system

Hardware	Operating System	Network Connection	Memory
PC	Windows XP	Dial-up	64MB
Laptop	Linux	DSL	128MB
PDA	BeOs	Cable	256MB

Interaction testing can offer significant savings. Indeed a system with 20 factors and 5 levels each would require $5^{20} = 95,367,431,640,625$ exhaustive test configurations.

Interaction testing techniques can be applied for interactions of $t > 2$ (factor,value) selections; t is the *strength* of the test. For pairwise testing, $t = 2$. An appropriate value for t to provide adequate coverage depends on the complexity of the system under test, and is not known. Initial research by Kuhn *et al.* [22, 23] indicates that choosing $2 \leq t \leq 6$ can be effective. Kuhn [22] reports in one study that more than 70% of defects were caught with 2-way coverage; ap-

Table 2: A small interaction test suite

Test No.	H/ware	Operating System	Network	Memory
1	PC	Windows XP	Dial-up	64MB
2	PC	Linux	DSL	128MB
3	PC	BeOS	Cable	256MB
4	Laptop	Windows XP	Cable	128MB
5	Laptop	Linux	Dial-up	256MB
6	Laptop	BeOS	DSL	64MB
7	PDA	Windows XP	DSL	256MB
8	PDA	Linux	Cable	64MB
9	PDA	BeOS	Dial-up	128MB

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.
Copyright 2005 ACM 1-58113-963-2/05/0005 ...$5.00.

proximately 90% with 3-way coverage; and 95% with 4-way coverage. In another study, 109 software-controlled medical devices recalled by the US Food and Drug Administration (FDA) are examined [23]. Pair-wise testing uncovered 97% of the flaws. Only three devices required coverage higher than two.

Dunietz *et al.* relate interaction coverage to code coverage; they suggest that t-way interaction coverage for "small" t reduces redundancy in block coverage while still identifying many faults [15]. High code coverage from interaction testing is also achieved by Burr *et al.* [1]. Yilmaz *et al.* [33] use interaction testing to provide fault localization. In these studies, interaction testing provides a significant reduction in the number of tests while succeeding in identifying and isolating faults [1, 15, 33].

Interaction testing is not new. In 1926, Fisher [16] pioneered interaction tests in agricultural experiments, assessing the contributions of different fertilizers to crop yield in the context of soil heterogeneity and environmental factors such as erosion, sun coverage, rainfall, and pollen. Testing was limited by available resources. It might have taken hundreds of years to test exhaustively. Fisher applied interaction testing so that every pair of factors affecting the yield was included exactly once. Since 1926, interaction testing has been widely used. For instance, interaction testing has been applied to combinatorial high throughput experiments (CHTE) [2], to test interactions that occur in cell signaling pathways [26], and to software testing [4, 6, 22, 23].

The underlying combinatorial object in all of these is a "mixed level covering array", defined formally later. Available algorithms for the construction of covering arrays arise from many different sources [10]. Combinatorial methods can offer fast constructions; they rely on the existence of specific algebraic or combinatorial objects [10, 18]. For example, TConfig [30, 31, 32] develops a recursive construction method based on orthogonal arrays. AETG [5] and CTS [19] can use a library of combinatorial objects, and CTS in addition employs combinatorial constructions. See also [21]. Until this time, the most severe limitation is that the combinatorial tools are most appropriate for a restricted set of parameters. If the time to generate a covering array is of utmost concern, and the sizes of the problems fit within a specific range of parameters, then algebraic methods address this well. However, when large ranges of parameters arise, more general techniques are needed.

Exact algorithms (integer programming and backtracking) have met success only for very small problems. Heuristic search has been more effective. Simulated Annealing (SA) [8, 9] and Tabu search [25] have produced many of the smallest available covering arrays. In these approaches, transformations are repeatedly applied to the current putative covering array, and a transformation is "accepted" using an heuristic decision rule (see [7, 8, 9, 25] for details).

If the size of the test suite is the overriding concern, simulated annealing or tabu search often yields the best results. However the substantial time required (and perhaps the complexity of implementing such methods) has led to the widespread use of simpler heuristics, such as hill-climbing [8] and greedy methods. See [8, 25] for data on accuracy and execution time of simulated annealing and tabu search. It may also be undesirable that randomization without the use of stored seeds in these types of methods produce different results each time the method is used.

Many algorithms for the construction of covering arrays are greedy, and generate one test at a time. Some examples of these algorithms include AETG [4, 5, 6], DDA [11], and TCG [28]. Combinatorial algorithms for generating designs and coverings one row at a time have a long history. Wells [29] discussed this strategy thirty years ago, and it was used to great benefit by Gibbons [13, 17]. Greedy variants have been treated in [12, 20], but the first such method for covering arrays was AETG [5].

In this paper, we focus on generalizing one-test-at-a-time greedy algorithms because they appear to provide general, fast techniques for the construction of test suites; our focus is therefore on the size of a test suite that can be constructed rapidly. This exploration does not encompass all greedy methods; IPO [27], for example, adjoins both new factors and new tests using a different greedy strategy. See also [3, 14].

We begin with some formal definitions. A *covering array*, $CA_\lambda(N;t,k,v)$, is an $N \times k$ array. In every $N \times t$ subarray, each t-tuple occurs at least λ times. In our application, t is the *strength* of the coverage of interactions, k is the number of components (factors), and v is the number of symbols for each component (levels). In all of our discussions, we treat only the case when $\lambda = 1$, i.e. that every t-tuple must be covered at least once.

This combinatorial object is fundamental when all factors have an equal number of levels. However, software systems are typically not composed of components *(factors)* that each have exactly the same number of parameters *(levels).* Then the mixed-level covering array can be used. A *mixed level covering array*, $MCA_\lambda(N;t,k,(v_1,v_2,\ldots,v_k))$, is an $N \times k$ array. Let $\{i_1,\ldots,i_t\} \subseteq \{1,\ldots,k\}$, and consider the subarray of size $N \times t$ obtained by selecting columns $i_1,\ldots,i_t$ of the MCA. There are $\prod_{i=1}^{t} v_i$ distinct t-tuples that could appear as rows, and an MCA requires that each appear at least once. We use an "exponential" notation for the collection $(v_i : i = 1,\ldots,k)$ of numbers of levels: $g_1^{u_1} \cdots g_s^{u_s}$ is used to represent a tuple of $\sum_{i=1}^{s} u_i$ numbers, of which u_i are equal to g_i for $1 \leq i \leq s$.

2. FRAMEWORK OF GREEDY METHODS

Greedy algorithms [4, 6, 11, 28] have been proposed to obtain competitive results in accuracy, efficiency, and consistency. Adaptability for *seeds* (required tests) and *avoids* (forbidden tests) has also been examined. Published methods work well on different types of data. While a framework can be used to provide a taxonomy of the several known methods [7], our interest extends beyond these specific methods. We do not propose yet another greedy algorithm, but rather define and study a framework of several thousand instantiations giving greedy methods. The framework is therefore studied to identify the impact of fundamental decisions in its instantiation.

The generation of covering arrays and mixed-level covering arrays that fall within the greedy framework can be described at a high level. The overall goal in constructing a covering array is to create a 2-dimensional array in which all t-tuples associated with a specified input are covered. Using a greedy approach, this collection is built one row at a time by fixing each factor with a level value. The order in which factors are fixed may vary, as can the scheme for choosing levels (values that can be selected for a factor). A row may be selected from multiple candidates. When more than one

```
set $MinArray$ to $\infty$
// Layer One - Repetition decision
repeat $\boxed{RepetitionCount}$ times
  start with no rows in $C$
  $N$=0
  while there are uncovered $t$-tuples in $C$
    // Layer Two - Candidate decision
    repeat $\boxed{CandidateCount}$ times
      start with an empty test (row) $R$
      set $Best = 0$
      while free factors remain
        // Layer Three - Factor ordering decisions
        rank all free factors according to a
            $\boxed{\text{factor ordering selection criterion}}$
        among factors tied for best, select a subset $T$
            using a $\boxed{\text{first factor tie-break}}$
        among factors in $T$, select a subset $F$
            using a $\boxed{\text{second factor tie-break}}$
        // Layer Four - Level selection decisions
        let $f$ be the lexicographically first factor in $F$
        rank all possible values for $f$ in $R$ using
            a $\boxed{\text{level selection criterion}}$
        among all best values for $f$, select a subset $V$
            using a $\boxed{\text{first level tie-break}}$
        among values in $V$, select a subset $W$
            using a $\boxed{\text{second level tie-break}}$
        let $v$ be the lexicographically first value in $W$
        fix factor $f$ to value $v$ in test $R$
      end while
      If $R$ covers $\sigma > Best$ $t$-tuples uncovered
          in $C$, set $Best = \sigma$, $B = R$
    end repeat
    add row $B$ to $C$
    $N$++
  end while
  if $C$ has $N < MinArray$ rows, set $MinArray = N$
      and $BestArray = C$
end repeat
report $BestArray$
```

Figure 1: Framework Pseudocode

candidate is permitted, several rows are constructed and one is chosen to add to the covering array. Once all t-tuples have been covered, the covering array is complete. The goal is to construct the smallest covering array possible, so if random selections occur, the covering array can be regenerated numerous times to select the best result.

Let us look at a generic method of this type as shown in Figure 1. We generate one row at a time; at a given step, a row has some factors for which level values have been chosen (*fixed*), and the remaining factors *free.*

To instantiate this method, a number of decisions must be made, shown in boxes in the skeleton of pseudocode in Figure 1. The four major decision points (*layers* of the framework) are: *layer one - the number of repetitions; layer two - the number of test candidates; layer three - factor ordering (including tie-breaking); and layer four - level selection (including tie-breaking).* The specification of these four layers dominate the accuracy and efficiency of such algorithms. Naturally the framework could extend to further levels of tie-breaking.

2.1 Layer one – Repetitions

Covering arrays may be generated a number of times, keeping the smallest size. In the pseudocode of Figure 1, the variable *RepetitionCount* represents the number of these repetitions. Repetitions are only useful when some decision is made randomly. More repetitions require lengthier execution times, and consistency from one run to the next are not ensured.

2.2 Layer two – Candidates

An algorithm may generate a *CandidateCount* number of rows, choosing the best one to add to the covering array under construction.

2.3 Layer three – Factor ordering

There are infinitely many ways to specify *factor ordering selection criterion.* While the framework does support hybrid rules, we have chosen in this initial study to focus only on a few specific rules here. Factors may be ranked by *the number of levels associated with a factor; by number of uncovered pairs involving this factor and the fixed factors; by the expected number of pairs covered including both fixed and free factors (*density*); or randomly.* These are selected with two goals in mind: we treat only heuristics that select every factor by the same rule, and we examine ones motivated by (*not necessarily equivalent to*) rules used in the available algorithms.

TCG [28] orders factors in nonincreasing order by the number of associated levels. AETG [4, 6] selects the first factor as one having the most uncovered pairs left. Random factor ordering is used for the remainder. While this “hybrid” factor ordering rule may be more effective than using uncovered pairs or random ordering alone, we opt here to include rules that treat every factor in the same manner.

DDA [11] orders factors using a density formula that calculates the *expected number* of uncovered pairs. To make this precise, we define the *current factor* as the factor for which density is being calculated, $r_{i,j}$ as the number of uncovered pairs between the current factor i and another factor j, and λ_{max} as the largest cardinality of the factors. The density for factor i with respect to factor j is:

1. if both factors have more than one level involved in uncovered tuples left,
   $\delta = (\frac{r_{i,j}}{\lambda_{max}^2})^2$
2. if only one factor has one level left and the other has greater than one level remaining, then $\delta = (\frac{r_{i,j}}{\lambda_{max}})^2$
3. if both factors have exactly one level left, then if a new pair is covered, $\delta = 1.0$, otherwise, $\delta = 0.0$

The *density* of factor i is the summation of these local densities over each factor $j \neq i$.

Three of these factor ordering methods can suffer from ties. To break ties, one of the following tie breaking schemes may be used: *take lexicographically first, take one at random, or take one with the most uncovered pairs remaining.* Not all tie-breaking rules given succeed in completely resolving a tie; hence more rounds of tie-breaking may be necessary. In our implementation, tie-breaking is taken to only two levels which are referred to as *first factor tie-break* and *second factor tie-break* in the pseudocode of Figure 1. In the case that a tie moves beyond two levels, the tie is broken by selecting the factor that is lexicographically first.

2.4 Layer four – Level selection

Level selection attempts to maximize the number of pairs to be covered, or that are expected to be covered for a row. Again, rules for level selection are potentially infinite. We have chosen specific *level selection criterion*: *by the number of uncovered pairs involving this level (value) of the current factor and the fixed factors; by the expected number of pairs involving this level of the current factor covered including both fixed and free factors; or randomly.*

AETG [5, 4] and TCG [28] select based on the number of uncovered pairs to be covered. DDA [11] selects a level based on *density*, the *expected* number of pairs to be covered. This is calculated as before (tentatively) fixing the current factor to each possible level value in turn; the level leading to the largest reduction in density is chosen. Random level selection is also included in our experiments, although it is not represented in the published algorithms.

A level selection tie-break is needed. The following strategies are used: *take the lexicographically first, lexicographically last, random, least frequently used, least recently used, or based on the level involved in the most uncovered pairs.* In our implementation, we limit the depth of tie-breaking to *first level tie-break* and *second level tie-break.* If a tie-break is necessary beyond two levels, a deterministic rule is in place to select the lexicographically first.

3. FRAMEWORK EXPERIMENTS

Evidently many greedy algorithms can be developed within the framework for the construction of covering arrays (AETG, TCG, and DDA among them). We have not made an effort to represent the nuances of each of these methods; for example, AETG uses a hybrid factor ordering, and DDA allows a more general definition of density, and these are not represented in our first experiments with the framework. Our interest is to explore the impact of each of the decisions made in instantiating the framework to a specific algorithm. In doing so, we consider not only main effects of the individual decisions made, but also interactions among these decisions.

Results are reported on several data sets that were chosen to encompass a variety of situations. These include small, medium, and large inputs for both covering array and mixed-level covering arrays for $t = 2$. Scenarios are also included for mixed-level covering arrays that have larger variations in numbers of factors and levels. Given that $t = 2$ for all experiments run, we refer to the input by the level values only (e.g. $CA(2, 3^5)$ which has 5 factors each with three levels is written as input 3^5).

3.1 Layer Four – Level selection

When selecting a level, it is reasonable to choose one that offers the largest increase in covered pairs. Uncovered pairs selects values for levels that offer the largest increase of newly covered pairs in relation to fixed factors. Alternatively density examines the a chosen level with respect to both fixed and free factors. We examine three options for level selection: uncovered pairs, density, and random. Does level selection make a significant contribution towards the array size? Results are generated using over 5,000 instantiations of the framework. We use:

Table 3: Sizes using three different level selection rules

	Random Bst/Avg/Wst *(size)*	**Uncovered Pairs** Bst/Avg/Wst *(size)*	**Density** Bst/Avg/Wst *(size)*
6^4	55/131.58/269	43/115.41/205	39/43.47/49
$10^1 9^1 8^1$ $7^1 6^1 5^1$ $4^1 3^1 2^1 1^1$	136/328.1/596	97/2088.2/9163	90/95.74/108
$8^2 7^2 6^2 5^2$	113/248.01/485	74/509.78/1861	68/73.7/85
$6^6 5^5 3^4$	92/180.9/399	57/125.66/369	55/60/69

Layer 1 (Repetitions) - 1, 10

Layer 2 (Candidates) - 1, 5, 10

Layer 3 (Factor ordering selection) - random, uncovered pairs, and density

Layer 3 Tie-breaking(to two levels)- take first, most pairs left, and random

Layer 4 (Level selection) - uncovered pairs, density, and random

Layer 4 Tie-breaking(to two levels) - take first, take least frequent, least recently used, most pairs left, take last, and random

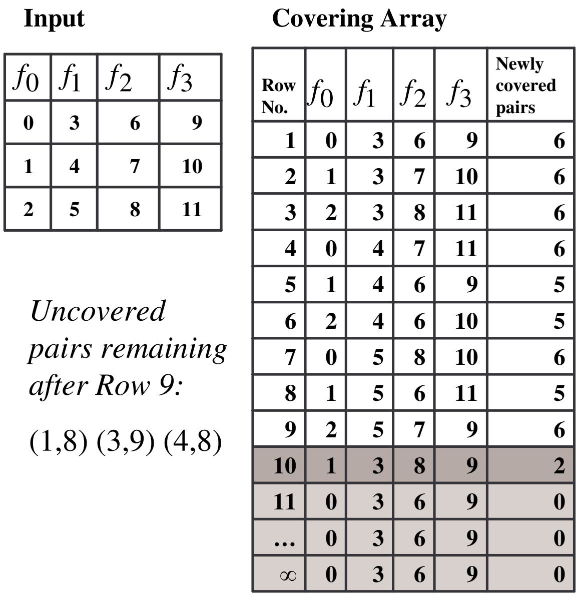
A standard ANOVA analysis is a well known statistical technique for measuring whether or not a significant relation exists among variables [24]. Each combination of framework settings described above was one test that was measured by the response variable of the *size* of the covering array generated. All of these combinations of framework settings and responses were used as input to the ANOVA test. The output of the ANOVA was then the contribution of each layer and their interactions towards size.

The ANOVA results of this experiment indicate that level selection is the main contributor to the array size produced; however a large percentage of the variance could not be accounted for. For instance, in covering arrays constructed using the input 6^4, level selection contributed 20.29% towards accuracy, and 54.74% could not be attributed to any layer or interaction between layers. A larger input, 3^{40}, produced similar results. Level selection contributed 26.42% towards accuracy, while 33.37% was unaccounted for.

Mixed level covering arrays show the same behavior. Using the input $10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$, level selection made the largest contribution of 45.93%; however, 23.58% could not be accounted for. Using input $8^2 7^2 6^2 5^2$, level selection contributed 48.27% and 21.57% was unaccounted for. In each example tested for mixed level covering arrays, the other layers contributed less than 1% towards accuracy.

Level selection overshadows all other framework layers, and a large amount of contribution cannot be attributed to any layer or interaction of layers. This is not surprising, since the three level selection rules exhibit quite different behaviors. We examine each in turn.

Results are given in Table 3. Best, average, and worst sizes are reported for each. Random level selection yields very poor results, occasionally producing rows that cover no new pairs. For instance, for input $6^6 5^5 3^4$, there is an enormous difference between the best size of 92 encountered and the worst size of 399. The primary influence on size is the



Input

f_0	f_1	f_2	f_3
0	3	6	9
1	4	7	10
2	5	8	11

Uncovered pairs remaining after Row 9:

(1,8) (3,9) (4,8)

Covering Array

Row No.	f_0	f_1	f_2	f_3	Newly covered pairs
1	0	3	6	9	6
2	1	3	7	10	6
3	2	3	8	11	6
4	0	4	7	11	6
5	1	4	6	9	5
6	2	4	6	10	5
7	0	5	8	10	6
8	1	5	6	11	5
9	2	5	7	9	6
10	1	3	8	9	2
11	0	3	6	9	0
...	0	3	6	9	0
∞	0	3	6	9	0

Figure 2: Partial generation of input 3^4.

random selections made, and these introduce a substantial 'unexplained' variance in the statistical results. Although every covering array produced is finite, sizes on average are too large, and even in the best case sizes are not competitive.

A problem also arises with level selection based on uncovered pairs. To demonstrate this, consider the input in Figure 2 and the following instantiation of the framework:
Layer 1 (Repetitions) - 1
Layer 2 (Candidates) - 1
Layer 3 (Factor ordering selection) - number of levels
Layer 3 Tie-breaking - lexicographically first
Layer 4 (Level selection) - uncovered pairs
Layer 4 Tie-breaking - lexicographically first

Figure 2 shows a pathological example of undesirable behavior. All factors have exactly three levels so the algorithm orders the factors lexicographically: f_0, f_1, f_2, f_3. Factors are fixed one at a time by selecting the level that adds the largest number of newly covered pairs in relation to fixed factors. The generation of the first 10 rows is straightforward. When generating the 11th row (shown in Figure 2), there is one pair left to be covered, (4,8). The algorithm begins by selecting a level value for f_0 that covers the most new pairs. Factor f_0 is not involved in any remaining uncovered pairs, so there is a tie among all levels. The lexicographically first is 0. Next, factor f_1 is assigned a level value by choosing the one with the largest increase in newly covered pairs in relation to fixed factor, f_0. Since (f_0, f_1) have no uncovered pairs remaining, tie-breaking selects the level value 3. The level for f_2 is chosen to maximize the number of newly covered pairs in relation to fixed factors f_0 and f_1. However, f_2 is not involved in any uncovered pairs with f_0 at 0 or f_1 at 3. Tie-breaking selects the level value 6. Finally, f_3 also cannot be selected to cover new pairs in relation to the fixed factors. Tie-breaking selects the level value 9. The row constructed covers *no new pairs*! Every subsequent row makes the same decisions.

Indeed, methods that use level-based factor ordering and choose these values only in relation to fixed factors may encounter this problem. Alternatively, if level selection had looked ahead to the free factors, or if the first factor had been chosen to be in at least one uncovered pair, we would not encounter the same problem. The level for f_1 could have been selected instead to consider that selecting the level value 4 would have allowed the future factor, f_2, to be fixed to a value covering a new pair.

Table 4: Standard ANOVA results of the top three layers of the framework

Input	*% Contribution of Layers towards Size (Layer 1/ 2/ 3)*
$10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$	.85/ .47/ 73.18
$8^2 7^2 6^2 5^2$	1.06/ 1.07/ 70.86
$6^6 5^5 3^4$	1.16/ .84/ 69.43
3^4	5.7/ .092/ 10.72
6^4	4.8/ .89/ 2.47
3^{40}	10.47/ .05/ 28.84

Problems with level selection based on uncovered pairs occur with both deterministic and random implementations. Since only the i fixed factors are included in the calculations for level selection, it is possible to prevent the coverage of pairs involving free factors. This can result in overly large covering arrays as shown in Table 3, or in infinite loops as in Figure 2.

Based on the data in Table 3, density based level selection is the best choice overall. The ANOVA results reported earlier indicate that level selection is dominant, when these three choices are permitted. However, the results for the three separately establish that they exhibit very different characteristics in terms of array size. Moreover, random level selection, and the incompatibility between factor and uncovered pairs level selection methods, account for the unexplained variance discussed earlier. For this reason, in order to proceed we henceforth do level selection using density. This is emphatically not to say that the other level selection rules cannot produce competitive results; our experiments are constrained to speak only about the specific rules implemented, and many other rules are possible. A more thorough analysis is underway to draw more definitive conclusions about level selection methods.

Having fixed level selection to be density based, are there dominant layers within the greedy framework? To address this, several data sets were run through the framework using over 5,000 variations of settings each. These settings include:
Layer 1 (Repetitions) - 1, 10
Layer 2 (Candidates) - 1, 5, 10
Layer 3 (Factor ordering selection) - levels, random, uncovered pairs, density
Layer 3 Tie-breaking (to two levels)- take first, most pairs left, and random
Layer 4 (Level selection) - density
Layer 4 Tie-breaking (to two levels) - take first, take least frequent, least recently used, most pairs left, take last, and random

In this experiment, a variety of rules were used for the top three layers of the framework while the level selection setting was set only to density. Using these settings, a standard ANOVA analysis identified the contribution of the top

Table 5: Standard ANOVA results - Percent contributions of feature sets towards size using density based level selection

	$10^1 9^1 8^1 7^1 6^1$ $5^1 4^1 3^1 2^1 1^1$	$8^2 7^2 6^2 5^2$	$6^6 5^5 3^4$	3^4	6^4	3^{40}
Repetitions	-	1.0645	1.157	5.698	4.797	10.466
Candidates	-	1.0656	-	-	-	-
Factor ordering selection	73.176	70.8598	69.431	10.723	2.472	28.836
Tie-breaking (TB)						
Factor ordering selection Primary tie-break (TB1)	-	-	-	2.16	-	1.288
Factor ordering selection Secondary tie-break (TB2)	-	-	-	-	-	-
Level selection Primary tie-break (TB1)	4.526	1.0975	4.148	33.646	21.267	3.433
Level selection Secondary tie-break (TB2)	-	-	-	-	-	-
Interactions between layers						
Repetitions and Candidates	-	1.4686	-	-	1.956	-
Candidates and Factor ordering selection	1.827	2.2132	1.061	1.45	-	-
Factor ordering selection and its TB1	-	-	-	3.435	1.29	2.866
Factor ordering selection and Level selection TB1	3.535	2.1676	6.285	3.157	18.161	3.604
Lack Of Fit	13.649	17.7303	14.41	35.786	44.196	44.774
**Values that contribute <1% are not reported*						

three framework layers towards array size. Based on a variety of input sets, the order of layers with highest dominance to least is typically: Layer 3 – Factor ordering; Layer 1 – Repetitions; Layer 2 – Candidates. See Table 4. A more detailed ANOVA analysis is given in Table 5. The percentage of each feature's contributions towards size is shown, along with the lack of fit. All contributions smaller than 1% are suppressed in the table.

The first three columns in Table 5 show the results for three mixed level covering arrays. The dominant feature in each of these is factor ordering. For instance, for $8^2 7^2 6^2 5^2$, approximately 70% of contributions towards size are accounted for by the factor ordering rules. The last three columns show the results for three fixed level covering arrays. In these cases, first level tie-break(TB1) is much more important; factor ordering and repetitions remain significant. Tie-breaking has no significant impact beyond the first level; this justifies, after the fact, our decision in the framework specification to limit variations in tie-breaking to two levels. Other test cases may, of course, show that second level tie-breaking can be significant.

Table 5 shows that in 5 out of 6 cases, factor ordering is the most influential selection. The next most significant feature is first level tie-break. Third is the interaction of these. Finally, repetitions is the fourth most significant.

3.2 Layer three – Factor ordering

According to these results, factor ordering is one of the most important decisions to make. In Table 5, factor ordering contributed up to 73% towards size. How much variation is found among factor ordering rules? In this experiment, we select the following:

Layer 1 (Repetitions) - 1
Layer 2 (Candidates) - 1
Layer 3 (Factor ordering selection) - levels, number of uncovered pairs, density, and random
Layer 3 Tie-breaking(to two levels) - take first, most pairs left, random
Layer 4 (Level selection) - density
Layer 4 Tie-breaking (to two levels) - take first, least frequent, least recently used, most pairs left, least recently used, random, and take last

The best, average, and worst results are partitioned by factor ordering methods in Table 6. For instance, the first column shows the best, average, and worst cases encountered for all of the settings in the experiment that ordered factors based on their number of levels. The first three columns that include factor ordering based on levels, density, and uncovered pairs attain similar results. Only random factor ordering, shown in the last column of the table, is less viable here. Random ordering produces the worst results in half of the examples. The other factor ordering rules sometimes outperform each other.

We omit random factor ordering, and consider further parameters for covering arrays. The framework settings are as before, with the omission of random factor ordering. As shown in Table 7, density appears to be the best choice among the rules considered. Factor ordering based on levels had poor performance in these examples when there were large numbers of factors with few different numbers of levels. In this case, ordering by number of levels is rather arbitrary. Factor ordering based on the numbers of uncovered pairs works better than ordering based on numbers of levels in these examples. However, the results are still not competitive with density. Again, we caution the reader that other factor orderings, in particular hybrid ordering schemes not treated here, may well be competitive.

3.3 Layer two – Candidates

What contribution can multiple candidates offer? In Table 5, candidates contributed from of .05% to 1.07% towards array size. Hence, while maintaining multiple candidates is relatively time-consuming, they do not account for a substantial variance in array size in our experiments. Here we look at a collection of individual runs to evaluate impacts of multiple candidates.

Variations in numbers of candidates were studied using settings of 1, 5, and 10 candidates. The instantiations of

Table 6: Sizes using four factor ordering rules

Factor Ordering	Levels *(size)*	Density *(size)*	Uncovered pairs *(size)*	Random *(size)*
3^4				
Best	9	9	9	9
Avg	10.78	11.96	11.36	12.46
Worst	24	27	25	25
Stdev	3.17	3.63	2.78	2.62
6^4				
Best	41	41	42	43
Avg	43.68	43.59	44.03	45.19
Worst	49	48	47	49
Stdev	2.41	1.37	1.62	1.33
10^4				
Best	115	116	115	117
Avg	123.55	119.95	120.19	121.32
Worst	130	125	125	126
Stdev	2.84	1.69	1.88	1.99
$10^1 9^1 8^1 7^1 6^1$ $5^1 4^1 3^1 2^1 1^1$				
Best	90	90	90	93
Avg	93.42	92.99	93.17	99
Worst	95	97	96	108
Stdev	1.21	1.35	1.27	2.3
$8^2 7^2 6^2 5^2$				
Best	72	69	69	76
Avg	73.75	72.54	72.33	80.13
Worst	77	76	77	85
Stdev	1	1.23	1.37	1.81
$6^6 5^5 3^4$				
Best	57	55	55	58
Avg	61.04	57.42	57.81	61.45
Worst	63	61	61	66
Stdev	1.59	0.9	1.02	1.3
$3^4 4^5$				
Best	24	23	22	23
Avg	26.95	24.06	24	25.38
Worst	30	26	28	29
Stdev	1.03	0.83	0.72	0.95
3^{13}				
Best	19	18	18	18
Avg	20.22	19.26	19.66	19.98
Worst	22	21	21	22
Stdev	0.58	0.48	0.55	0.72
3^{40}				
Best	26	25	25	25
Avg	27.25	26.19	26.42	26.66
Worst	28	28	27	29
Stdev	0.51	0.53	0.55	0.65

Table 7: Sizes using three factor ordering rules on more diverse data sets

Factor Order-ing	Levels Best/Avg/ Worst *(size)*	Density Best/Avg/ Worst *(size)*	Uncovered pairs Best/Avg/ Worst *(size)*
$7^8 2^{20}$	82/85.09/86	78/78/78	76/78.45/79
$5^{10} 2^{10}$	48/49.64/50	45/46/47	46/46.64/47
$3^{50} 2^{50}$	29/29.82/30	27/28/29	28/28.82/29
2^{80}	14/14.82/15	13/13.41/12	14/14.18/15
3^{100}	32/32.82/33	31/31.73/32	32/32/32

Table 8: Sizes using multiple candidates

	1 Cand Best/Avg/ Worst *(size)*	5 Cand Best/Avg/ Worst *(size)*	10 Cand Best/Avg/ Worst *(size)*
3^4	9/ 10.3/ 15	9/ 10.2/ 13	9/ 10.2/ 15
6^4	39/ 43.7/ 49	40/ 43.4/ 48	40/ 43.3/ 48
10^4	112/120.4/130	112/119.2/126	112/119/127
$10^1 9^1 8^1$ $7^1 6^1 5^1$ $4^1 3^1 2^1$	90/95.23/108	90/94.8/106	90/94.71/103
$8^2 7^2 6^2 5^2$	69/ 74.11/ 85	68/ 73.52/ 85	68/73.55/85
$6^6 5^5 3^4$	55/ 59.84/ 66	55/ 59.46/ 64	55/59.36/ 65
$3^4 4^5$	23/ 25.24/ 30	23/ 25.22/ 30	22/25.13/ 29
$5^1 3^8 2^2$	20/ 21.77/ 25	19/ 21.48/ 25	19/21.58/25
$5^1 3^8 2^2$	17/ 19.52/ 22	17/19.42/ 22	17/19.42/22
3^{40}	25/26.62/29	25/26.39/28	25/26.37/29

the framework are:
Layer 1 (Repetitions) - 1, 10
Layer 2 (Candidates) - 1, 5, 10
Layer 3 (Factor ordering selection) - levels, density, uncovered pairs and random
Layer 3 Tie-breaking (to two levels) - take first, most pairs left, random
Layer 4 (Level selection) - Density
Layer 4 Tie-breaking (to two levels) - take first, take least frequent, least recently used, most pairs left, take last, and random

As shown in Table 8, multiple candidates only offered slight improvements. However, increased candidates also did not significantly degrade the worst cases encountered.

3.4 Layer one – Repetitions

Repetitions can be used in algorithms that have elements of randomness. The best, average, and worst cases of covering array sizes encountered by such algorithms can vary from run to run. In Table 5, repetitions contributed in the range of .85% to 10.47%. In this experiment, small to large numbers of repetitions are explored using 164 instantiations of the greedy framework. The framework settings are:
Layer 1 (Repetitions) - 1, 50, 1,000, 10,000, 20,0000
Layer 2 (Candidates) - 1
Layer 3 (Factor ordering selection) - levels, density, and random

Table 9: Sizes using multiple repetitions

Input	Number of Reps	Best	Avg	Worst
		(size)	*(size)*	*(size)*
6^4	1	46	46.333	47
	10	42	46.033	50
	50	43	46.013	52
	100	42	45.94	52
	1,000	41	45.92	52
	10,000	41	45.895	52
	20,000	40	45.9	54
$8^2 7^2 6^2 5^2$	1	73	75.333	80
	10	70	74.967	84
	50	71	75.387	85
	100	70	75.32	85
	1,000	69	75.523	86
	10,000	68	75.526	87
	20,000	68	75.503	88
$6^6 5^5 3^4$	1	57	60	62
	10	56	60.433	66
	50	56	58.333	61
	100	56	60.7	67
	1,000	56	57.667	60
	10,000	55	60.69	69
	20,000	55	57	59

Layer 3 Tie-breaking - random
Layer 4 (Level selection) - density
Layer 4 Tie-breaking - random

Table 9 shows that more repetitions usually improve array size. Increased repetitions produced larger improvements than increased candidates did. There are at least two possible reasons why increased candidates are not performing as well. First, while there may be very many different covering arrays for the same parameters, there are often not as many unique candidates for each row. Second, multiple candidates correspond to a "steeper ascent" in the selection, making the method even more greedy. However there is no guarantee that a steepest ascent approach is the most accurate method to generate a covering array (see [8] for a discussion).

The data indicates that repetitions are often valuable. However these results may be confounded by factor ordering methods. We therefore look at the data again, partitioned by factor ordering rules. Results are averaged across the inputs and the difference between one repetition to 50, 1,000, 10,000, and 20,000 repetitions are reported in Table 10. Repetitions improve all three factor-ordering methods; see Table 10. Algorithms with random factor ordering benefit the most from increased repetitions, followed by the ordering based on levels, and finally by density.

Increased repetitions improve accuracy at the cost of time. Repeating the algorithm n times will result in an execution time of approximately n times longer than one repetition. For instance, consider the data shown in Table 9 for $10^1 9^1 8^1 7^1 6^1 5^1 4^1 3^1 2^1 1^1$. The table shows that on average, the increase in repetitions improved accuracy. However, the run time also increased from an average of .1 second for one repetition to .95 second for ten repetitions, 9.48 seconds for 100 repetitions, and 93.07 seconds for 1,000 repetitions. In any practical application, a trade-off must be made between the emphasis on array size and that on the time to generate the test suites.

Table 10: Results of increased repetitions (partitioned by factor ordering rules)

	50 reps	1000 reps	10000 reps	20000 reps
	No. of rows reduced	*No. of rows reduced*	*No. of rows reduced*	*No. of rows reduced*
Levels	2.5	3.67	3.83	4.17
Density	1.83	3.17	3.67	3.5
Random	2.17	4	4.5	4.67

4. CONCLUSIONS

Algorithms for constructing covering arrays have been developed to meet practical software testing concerns. However, none has outperformed each of the others with respect to accuracy, consistency, efficiency, and extensibility. This paper introduced a framework for greedy algorithms that construct covering arrays and mixed-level covering arrays one row at a time, since these offer attractive methods for speed and reasonable accuracy. A four layer framework is introduced and empirically studied. This framework enables the development of statistical bases for the relative importance of each decision made in its instantiation as an algorithm.

The several thousand instantiations of this framework offer a variety of methods for software testers. In this framework, we initially saw that level selection is the most dominant layer. However, a closer look indicated that the results were obfuscated by some poor level selection rules that produce overly large results (in conjunction with our other possible selections). Without alternative rules for selecting other features, random and uncovered pairs level selections are not competitive. Density based level selection produces the best and most reliable results and were used in further studies to focus on the remaining three layers of the framework.

Once we restrict the level selection rule, factor ordering has the largest impact on size. Other choices are less significant. However, using multiple repetitions and multiple candidates lead to small improvements. When size is the dominant concern, an algorithm should give priority to multiple repetitions over multiple candidates. If the time to generate is the primary concern, then multiple repetitions and candidates should be avoided and good level and factor orderings are important to focus on.

This is admittedly a preliminary study. Despite many thousands of instantiations being considered, variants of factor and level selection techniques not treated here may produce competitive or better results. In particular, hybrid rules that make factor or level selections depending upon the specific factor or level being selected offer promise that is not explored here. The extension to higher strength has not been explored here, and is one topic of ongoing work. Our current work also concerns adding to the factor and level selection rules, in order to draw conclusions more generally. The impact of allowing more candidates, or more repetitions, is under investigation as well. In addition, here we have reported limited data concerning execution times, and this is of paramount concern given our stated motivation for examining greedy methods. Despite the need for a more comprehensive set of rules to populate our framework, it is important that the framework itself provides not only a mechanism for describing and developing these many variants, but also a means to make quantitative statements about the statistical significance of each decision made.

5. ACKNOWLEDGMENTS

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

6. REFERENCES

[1] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation, and code coverage. *Proceedings of the Intl. Conf. on Software Testing Analysis and Review*, pages 503–513, October 1998.

[2] J. N. Cawse. Experimental design for combinatorial and high throughput materials development. *GE Global Research Technical Report*, 29(9):769–781, November 2002.

[3] C. Cheng, A. Dumitrescu, and P. Schroeder. Generating small combinatorial test suites to cover input-output relationships. *Proceedings of the Third International Conference on Quality Software (QSIC '03)*, pages 76–82, 2003.

[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, October 1997.

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

[6] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):82–88, October 1996.

[7] M. B. Cohen. Designing test suites for software interaction testing. *Ph.D. Thesis, The University of Auckland, Department of Computer Science*, 2004.

[8] M. B. Cohen, C. J. Colbourn, P. B. Gibbons, and W. B. Mugridge. Constructing test suites for interaction testing. *Proc. Intl. Conf. on Software Engineering (ICSE 2003)*, pages 38–48, 2003.

[9] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling. Constructing strength three covering arrays with augmented annealing. *Discrete Mathematics*, to appear.

[10] C. J. Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche (Catania)*, to appear.

[11] C. J. Colbourn, M. B. Cohen, and R. C. Turban. A deterministic density algorithm for pairwise interaction coverage. *Proc. of the IASTED Intl. Conference on Software Engineering*, pages 242–252, February 2004.

[12] J. H. Conway and N. J. A. Sloane. Lexicographic codes: error correcting codes from game theory. *IEEE Trans. Inform. Theory*, 32:337–348, 1986.

[13] P. C. Denny and P. B. Gibbons. Case studies and new results in combinatorial enumeration. *J. Combin. Des.*, 8:239–260, 2000.

[14] A. Dumitrescu. Efficient algorithms for generation of combinatorial covering suites. *Proc. 14-th Annual Intl. Symp. Algorithms and Computation (ISAAC '03), Lecture Notes in Computer Science*, pages 300–308, 2003.

[15] S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing. *Proc. Intl. Conf. on Software Engineering (ICSE '97)*, pages 205–215, October 1997.

[16] R. A. Fisher. The arrangement of field experiments. *Journal of Ministry of Agriculture of Great Britain*, 33(9):503–513, November 1926.

[17] P. B. Gibbons, R. A. Mathon, and D. G. Corneil. Computing techniques for the construction and analysis of block designs. *Utilitas Math.*, 11:161–192, 1977.

[18] A. Hartman. Software and hardware testing using combinatorial covering suites. *Haifa Workshop on Interdisciplinary Applications of Graph Theory, Combinatorics, and Algorithms*, June 2002.

[19] A. Hartman and L. Raskin. Problems and algorithms for covering arrays. *Discrete Math.*, 284:149–156, 2004.

[20] A. Hartman and Z. Yehudai. Greedesigns. *Ars Combin.*, 29C:69–76, 1990.

[21] N. Kobayashi, T. Tsuchiya, and T. Kikuno. A new method for constructing pair-wise covering designs for software testing. *Information Processing Letters*, 81:85–91, 2002.

[22] 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*, pages 91–95, October 2002.

[23] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Engineering*, 30(6):418–421, October 2004.

[24] D. Montgomery. *Design and Analysis of Experiments 5th edition.* John Wiley and Sons, New York NY, 2001.

[25] K. Nurmela. Upper bounds for covering arrays by tabu search. *Discrete Applied Math.*, 138(9):143–152, March 2004.

[26] D. Shasha, A. Kouranov, L. Lejay, M. Chou, and G. Coruzzi. Using combinatorial design to study regulation by multiple input signals. a tool for parsimony in the post-genomics era. *Plant Physiology*, 27:1590–1594, 2001.

[27] K. Tai and L.Yu. A test generation strategy for pairwise testing. *IEEE Transactions on Software Engineering*, 28:109–111, 2002.

[28] Y. Tung and W. Aldiwan. Automating test case generation for the new generation mission software system. *IEEE Aerospace Conf.*, pages 431–37, 2000.

[29] M. B. Wells. *Elements of Combinatorial Computing.* Pergamon Press, Oxford-New York-Toronto, 1971.

[30] A. W. Williams. Determination of test configurations for pair-wise interaction coverage. Testing of communicating systems: Tools and techniques. *13th International Conference on Testing Communicating Systems*, pages 59–74, October 2000.

[31] A. W. Williams and R. L. Probert. A practical strategy for testing pair-wise coverage of network interfaces. *Seventh Intl. Symp. on Software Reliability Engineering*, pages 246–254, 1996.

[32] A. W. Williams and R. L. Probert. A measure for component interaction test coverage. *Proc. ACS/IEEE Intl. Conf. on Computer Systems and Applications*, pages 301–311, October 2001.

[33] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Intl. Symp. on Software Testing and Analysis*, pages 45–54, July 2004.