

Constructing Interaction Test Suites with Greedy Algorithms

Renée C. Bryce
Computer Science and Engineering
Arizona State University
Tempe, Arizona 85287-8809
rcbryce@asu.edu

Advisor: Charles J. Colbourn
Computer Science and Engineering
Arizona State University
Tempe, Arizona 85287-8809
colbourn@asu.edu

ABSTRACT

Combinatorial approaches to testing are used in several fields, and have recently gained momentum in the field of software testing through *software interaction testing*. One-test-at-a-time greedy algorithms are used to automatically construct such test suites. This paper discusses basic criteria of why greedy algorithms have been appropriate for this test generation problem in the past and then expands upon how greedy algorithms can be utilized to address test suite prioritization.

Categories and Subject Descriptors

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

General Terms

Algorithms, Measurement, Experimentation

Keywords

Biased covering arrays, covering arrays, greedy algorithm, mixed-level covering arrays, t-way interaction coverage, software interaction testing

1. INTRODUCTION

Software systems are complex and offer numerous configurations. Exhaustive testing of every configuration is often not possible due to limited testing resources. Software testers try to catch as many problems as possible within a prescribed testing budget. Software interaction testing complements current testing methodologies with economical test suites that cover all *t*-way interactions in a system. Many tests within these interaction test suites may be easily automated while others may require human intervention and incur set-up costs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE'05, November 7–11, 2005, Long Beach, California, USA.
Copyright 2005 ACM 1-58113-993-4/05/0011 ...\$5.00.

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

Table 2: Pairwise testing requires only 9 tests

Test No.	Hardware	Operating System	Network Connection	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

Software interaction testing uses fewer tests than exhaustive testing. Preliminary studies have indicated that the method is promising. For instance, Kuhn et al. study defects in 109 software-controlled medical devices that were recalled by the US Food and Drug Administration (FDA) [7]. They identified 97% of the defects with 2-way coverage. See [4] for other empirical studies of software interaction testing.

1.1 A software interaction test suite example

Consider the example of a modular software system where components interact as shown in Table 1. Four components each have three possible settings. Testing each configuration adds to the cost of testing. Exhaustively testing all combinations of settings requires $3^4 = 81$ combinations. Pair-wise interaction testing, as shown in Table 2, requires only 9 tests. For this example, every option for hardware has been combined with every option for operating system, network connection, and memory configuration. Every pair of options between operating system, network connection, and memory configuration have also been combined in at least one test.

1.2 Definitions

Behind the scenes of interaction testing, a combinatorial

object called a covering array is constructed. The covering array, $CA_\lambda(N; t, k, v)$, is an $N \times k$ array in which all n -tuples of size t occur at least λ times. In this application, t denotes to strength of coverage of interactions, k denotes the number of components, and v denotes the number of symbols for each component.

In the cases that not all components have the exact same number of possible settings, a mixed level covering array must be constructed. A *mixed level* covering array, $MCA(N; t, k, (v_1, v_2, \dots, v_k))$, is an $N \times k$ array on v symbols, where $v = \sum_{i=1}^k v_i$, with the following properties:

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

1.3 Benefits of Greedy Algorithms

Software interaction test suites can be generated using different types of algorithms. For instance, mathematical constructions, greedy algorithms, and heuristic search have been applied. While there is merit to all three of these types of algorithms, my work focuses primarily on greedy algorithms. The rationale for this is that greedy methods overcome two major obstacles that separately face mathematical constructions and heuristic search. Mathematical constructions work well on select inputs but no technique is yet known for accurate construction on a broad range of problem inputs. Heuristic search is removed from consideration as these methods report significant execution time.

Greedy algorithms provide reasonable sized test suites in relatively short amounts of time. They are also flexible. For instance, the initial software interaction testing paradigm assumed that all tests within the test suite were of equal importance. When prioritization of tests was introduced, greedy algorithms easily accommodated this for the special case of pair-wise coverage [2, 3]. Future work may extend this implementation to higher strength coverage, or even to nonuniform higher strength coverage. This is discussed in more detail shortly.

The remainder of the paper briefly summarizes our current research contributions and suggests future work. Section 2.1 summarizes a greedy framework for generating test suites, as well as, an extensive set of framework experiments for the special case of pair-wise coverage. Section 2.2 discusses research challenges that need to be addressed for greedy algorithms that generate higher strength test suites. Section 3.1 describes a current algorithm that prioritizes the ordering of tests within a test suite. Section 3.2 discusses adding nonuniform higher strength coverage as a further measure of prioritization.

2. GREEDY ALGORITHMS

Our current work [1] has defined a framework that identifies all of the decision points that one-test-at-a-time greedy algorithms for constructing covering arrays have in common. The framework has been extensively studied for the special case of pair-wise coverage. Future work will utilize this framework to develop greedy algorithms for higher strength coverage.

```

set MinArray to  $\infty$ 
repeat RepetitionCount times
  start with no tests (rows) in  $C$ 
   $N=0$ 
  while there are uncovered  $t$ -tuples in  $C$ 
    repeat CandidateCount times
      start with an empty test  $R$ 
      set  $Best = 0$ 
      while free factors remain
        rank all free factors according to a
          factor ordering selection criterion
        among factors tied for best, select a subset  $T$ 
          using a first factor tiebreak
        among factors in  $T$ , select a subset  $F$ 
          using a second factor tiebreak
        let  $f$  be the lexicographically first factor in  $F$ 
        rank all possible values for  $f$  in  $R$  using
          a level selection criterion
        among all best values for  $f$ , select a subset  $V$ 
          using a first level tiebreak
        among values in  $V$ , select a subset  $W$ 
          using a second level tiebreak
        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 test  $B$  to  $C$ 
     $N++$ 
  end while
  if  $C$  has  $N < MinArray$  tests, set  $MinArray = N$ 
    and  $BestArray = C$ 
end repeat
report BestArray

```

Figure 1: Framework Pseudocode

2.1 Current work: The Greedy Framework

The 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 an input are covered. This collection is built one test at a time by fixing each factor with a value. The order in which factors are fixed may vary (called factor ordering), as can the scheme for choosing levels (called level selection). A test may be selected from multiple candidates. When more than one candidate is permitted, several tests 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.

To instantiate this framework, a number of decisions must be made. The four major decision points highlighted in Figure 1 are: *the number of repetitions*; *the number of test candidates*; *factor ordering (including tiebreaking)*; and *level selection (including tiebreaking)*. The specification of these dominate the accuracy and efficiency of such algorithms.

Our current work [1] has instantiated several thousand

Table 3: ANOVA analysis - Percent contributions of feature sets towards size

Input:	$10^1 9^1 8^1 7^1 6^1$ $5^1 4^1 3^1 2^1 1^1$	$8^2 7^2 6^2 5^2$	$6^6 5^5 3^4$	3^4	6^4	3^{40}
Repetitions	-	1.0645	1.157	5.698	4.797	10.466
Candidates	-	1.0656	-	-	-	-
FO (Factor Ordering)	73.176	70.8598	69.431	10.723	2.472	28.836
FO Tiebreak 1	-	-	-	2.16	-	1.288
FO Tiebreak 2	-	-	-	-	-	-
LS Tiebreak 1	4.526	1.0975	4.148	33.646	21.267	3.433
LS Tiebreak 2	-	-	-	-	-	-
Interaction of Repetitions and Candidates	-	1.4686	-	-	1.956	-
Interaction of Candidates and FO	1.827	2.2132	1.061	1.45	-	-
Interaction of FO and FO Tiebreak 1	-	-	-	3.435	1.29	2.866
Interaction of FO and LS Tiebreak 1	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
<i>*Values that contribute <1% are not reported</i>						

algorithms that fall into this framework; each instantiation having a unique set of framework parameter settings. One experiment studied the following parameters:

- **Repetitions:** 1, 10
- **Candidates:** 1, 5, 10
- **Factor ordering (FO):** assign values to factors in order based on the number of levels associated with each factor, assign at random, assign in descending order of associated uncovered pairs, and in descending order based on density (defined formally in [5])
- **FO Tiebreak:** (to two levels): lexicographically first, most pairs left, and random
- **Level selection (LS):** assign levels to factors using density (defined formally in [5])
- **LS Tiebreak** (to two levels): lexicographically first, least frequent, least recently used, most pairs left, last, and random

For each run of these framework instantiations, the size of the solution was recorded. ANOVA testing then evaluated the impact of each framework parameter setting towards the response variable of size. This experiment was run on several data sets that had varying numbers of factors and levels. The top three most influential layers were typically: Factor Ordering; Repetitions; and Candidates. A more detailed ANOVA analysis is shown in Table 3. The percentage of each feature’s contributions towards size is shown. Contributions less than 1% are suppressed.

The first three columns in Table 3 show results for three mixed level covering arrays. The dominant feature in each is factor ordering. For instance, for $8^2 7^2 6^2 5^2$ (read as 2 factors with 8 levels, 2 factors with 7 levels, 2 factors with 6 levels, and 2 factors with 5 levels), approximately 70% of contributions are accounted for by the factor ordering rules. In 5 out of 6 cases, factor ordering is the most influential selection. The next most significant feature is first level tiebreak of level selection. Third is the interaction of these. Finally, repetitions is the fourth most significant. This experiment provides a highlight of work that is presented in more detail in [1].

The framework identifies the decision points that one-test-at-a-time greedy algorithms can define. Experiments provide a requisite of knowledge about the effects of different types of rules for the special case of pair-wise coverage. This information can be leveraged by tool designers. For instance, future developers may implement tools with new features for addressing constraints (prohibited combinations), for a new method of prioritization, or for higher strength coverage. The framework can help them to rapidly evaluate the decisions that can be made in their algorithms.

2.2 Future Work

We plan to develop a greedy algorithm for higher strength coverage since empirical study has indicated benefits of covering t-tuples of $t > 2$. (See [4, 7] for an examples). This work poses challenges algorithmically as new heuristics need to be developed, computational resources need to be managed, and a number of other practical concerns still need to be maintained. For instance, Table 4 shows a sample of inputs and the combinatorial growth of tuples that occur as the strength, t, increases. The input 3^{13} includes 702 pairs, 7,722 triples, and reaches over a million 6-tuples when $t=6$. As the size of the tuples and the numbers of them increase, the size of test suites grow. The number of computations and the execution time for the algorithms grow as well. My future research plans are to develop algorithms that are competitive in respect to size of test suites, execution time, and practical concerns.

3. PRIORITIZED TEST SUITES

Components of software systems may have varying priorities. Prioritization values may be derived directly from requirements or from criteria such as code coverage, cost estimate (i.e., how much execution time a test incurs), areas of recent changes, areas that testers believe to be particularly fault-prone, and others. (See [6] for further examples or prioritization criteria). Priorities may even change during the testing process. In any case, a tester may request a complete test suite and attempt to run as many tests as they can budget. In this context, it can be important to either test the most important items first or to cover only certain parts of a system with higher strength coverage.

Table 4: Growth of tuples with increasing values of strength (t)

	2-tuples	3-tuples	4-tuples	5-tuples	6-tuples	...	exhaustive
$10^{19}9^{18}8^{17}7^{16}6^{15}5^{14}4^{13}3^{12}2^{11}1^1$	1,320	18,150	157,773	902,055	3,416,930	...	3,628,800
10^4	600	4,000	10,000	-	-	-	10,000
3^{13}	702	7,722	57,915	312,741	1,250,954	...	1,594,323
11^{16}	14,520	745,360	26,646,620	703,470,768	1,301,758,600	...	45,949,729,863,572,200

3.1 Current work: Prioritized test ordering

Our current work implements prioritization for ordering of tests within a test suite. The covering array does not distinguish ordering of tests even though certain tests can be more desirable than others. A biased covering array is needed to distinguish such orderings.

A ℓ -biased covering array is a covering array $CA(N; 2, k, v)$ in which the first ℓ rows form tests whose total benefit is as large as possible. That is, no $CA(N'; 2, k, v)$ has ℓ rows that provide larger total benefit.

Although precise, this definition should be seen as a goal rather than a requirement. Finding an ℓ -biased covering array is NP-hard, even when all benefits for pairs are equal. Worse yet, the value of ℓ is rarely known in advance. For these reasons, we use the term *biased covering array* to mean a covering array in which the tests are ordered, and for every ℓ , the first ℓ tests yield a “large” total benefit.

To generate a biased covering array, consider the following approach from our work in [2]. Call the factors f_1, \dots, f_k . Suppose that, for each i , f_i has ℓ_i possible values $c_{i,1}, \dots, c_{i,\ell_i}$. For each $c_{i,j}$, we assume that a numerical value between 0 and 1 is available as a measure of importance, where higher values denote higher priority. Every value τ for f_i has an importance $t_{i,\tau}$. A *test* consists of an assignment to each factor f_i of a value τ_i with $1 \leq \tau_i \leq \ell_i$. The first task is to quantify the preference among the possible tests. In order to capture important interactions among *pairs* of choices, importance of pairs is defined by a weight, $0 \leq \omega_i \leq 1$, where 1 is the strongest weight. Specifically, the importance of choosing τ_i for f_i and τ_j for f_j together is $t_{i,\tau_i} t_{j,\tau_j}$.

The *benefit* of a test (in isolation) is $\sum_{i=1}^k \sum_{j=i+1}^k t_{i,\tau_i} t_{j,\tau_j}$. Every pair covered by the test contributes to the total benefit, according to the importance of the selections for the two values. However in general we are prepared to run many tests. Rather than adding the benefits of each test in the suite, we must account a benefit only when a pair of selections has not been treated in another test. Let us make this precise. Each of the pairs (τ_i, τ_j) covered in a test of the test suite may be covered for the first time by this test, or may have been covered by an earlier test as well. Its *incremental benefit* is $t_{i,\tau_i} t_{j,\tau_j}$ in the first case, and zero in the second. Then the incremental benefit of the test is the sum of the incremental benefits of the pairs that it contains. The total benefit of a test suite is the sum, over all tests in the suite, of the incremental benefit of the test.

3.2 Future Work

Prioritization for interaction test suites exists using weights described here. This work can be extended to include higher strength coverage of interactions for certain components of a system. For instance, part of a system may be known to be stable and pair-wise interaction testing may be sufficient. However, testers may want to perform higher strength interaction testing on other parts of a system under test. In this

case, no greedy algorithms have been published to adapt for prioritization of nonuniform strength coverage. My future work plans to explore this direction of prioritization.

4. CONCLUSIONS

The research presented here is founded on the study of greedy algorithms to generate pair-wise interaction test suites. While pair-wise coverage may be valuable, algorithmic work is needed for tools to support both higher strength coverage and prioritization. For instance, pair-wise testing may offer a subset of the benefit of exhaustive testing, but a strength of $t \geq 2$ can offer additional benefit. Further, some parts of a system may be more critical to test than others and hence a nonuniform strength of coverage across a system may be desired. Testers may also want certain interactions to be prioritized so that they are covered earlier within the test suite. Tools to automatically generate higher strength test suites with such prioritization can improve the software interaction testing methodology.

5. REFERENCES

- [1] R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A Framework of Greedy Methods for Constructing Interaction Tests. *Proc. of the 27th International Conference on Software Engineering (ICSE)*, pages 145–155, May 2005.
- [2] R. C. Bryce and C. J. Colbourn. Test Prioritization for Pairwise Coverage. *Proc. of the Workshop on Advances in Model-Based Software Testing (A-MOST)*, to appear, May 2005.
- [3] R. C. Bryce, C. J. Colbourn, and Y. Chen. Biased Covering Arrays for Progressive Ranking and Composition of Web Services. *International Journal of Simulation and Process Modelling*, to appear.
- [4] C. J. Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche (Catania)*, to appear.
- [5] 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.
- [6] S. Elbaum, G. Rothermel, S. Kanduri, and A. Malishevsky. Selecting a Cost-Effective Test Case Prioritization Technique. *Software Quality Journal*, 12(3):185–210, 2004.
- [7] 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.