Augmenting Simulated Annealing to Build Interaction Test Suites

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

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

Alan C.H. Ling
Dept. of Computer Science
University of Vermont
Burlington, Vermont 05045
aling@emba.uvm.edu

Abstract

Component based software development is prone to unexpected interaction faults. The goal is to test as many potential interactions as is feasible within time and budget constraints. Two combinatorial objects, the orthogonal array and the covering array, can be used to generate test suites that provide a guarantee for coverage of all t-sets of component interactions in the case when the testing of all interactions is not possible. Methods for construction of these types of test suites have focused on two main areas. The first is finding new algebraic constructions that produce smaller test suites. The second is refining computational search algorithms to find smaller test suites more quickly. In this paper we explore one method for constructing covering arrays of strength three that combines algebraic constructions with computational search. This method leverages the computational efficiency and optimality of size obtained through algebraic constructions while benefiting from the generality of a heuristic search. We present a few examples of specific constructions and provide some new bounds for some strength three covering arrays.

1. Introduction

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

In this paper we use the term *component* to represent any stand-alone portion of a software system. Each component is a *factor* affecting the correctness or performance of the software system. A component or factor can be either a software program such as an operating system, or network protocol, or it may be a hardware platform, i.e. a memory module or printer model. A *configuration* of the component is one possible value or state for that component. In the case of an operating system, each different operating system is a configuration; in a hardware environment, each printer model is a configuration. We assume that each component has already been tested thoroughly at the unit level. Our concern is with faults that occur due to interactions between various combinations of configurations. A product line architecture example of this problem follows.

Suppose we are designing a new integrated RAID controller for PCs. The specifications state that this controller runs on multiple operating systems, supports a variety of RAID levels and disk interfaces, and can be used with different amounts of embedded memory. The controller comes with software that allows the purchaser to customize it for their desired RAID environment. Table 1 shows a simplified example of this system. It has four components, RAID level, operating system, memory configuration and disk interface. Each of these has three configurations. The controller is designed to support RAID 0, RAID 1, and RAID 5, to run on Novell Netware, Linux and Windows XP, to

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

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

support 64,128 and 256 MB of memory and to be used with three hard disk interfaces, Ultra 160-SATA, Ultra 160-SCSI and Ultra 320-SCSI.

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

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

In this example there are $3^4 = 81$ possible interactions among the component configurations. Before releasing this product to market all 81 of these combinations should be tested to detect interaction faults. In this particular example it may be possible to run all 81 tests, but the problem grows too large very rapidly. Suppose there are 20 components. If two of these have four possible configurations, while the rest have only three, we have $4^2 \times 3^{18}$ or 6,198,727,824 possible interactions. Since many of these interactions may involve changing hardware components, or switching to different operating systems the of testing all of these is impossible.

When it is not possible to test all interactions one can use heuristics to choose which ones are to be tested. This requires a trade-off between the number of tests run and the thoroughness of testing. One method which has been proposed for providing a defined and repeatable set of tests arises from methods used in statistical design of experiments. This method guarantees a certain amount of *interaction coverage* in software systems [5, 6, 7, 8, 11, 12, 22, 23, 24, 25]. We define *interaction coverage* as the size of the interaction subsets guaranteed to be tested from among all possible component configurations. To be precise, if we select a set S of factors, and further select a configuration for each factor in S, we obtain a *partial assignment* of configurations to the factors. We say that such a partial assignment is *covered* if at least one test of the test suite agrees with the assignment of values to the factors of S. The interaction coverage is n*-way* if for every subset S of at most n factors, every assignment of values to these factors is covered. Two-way coverage is often referred to as *pairwise* coverage. Similarly we often use *triples* to indicate 'three-way'.

Mandl was the first to use statistical design of experiments for testing compiler software [15]. Brownlie *et al.* extended this further for an internal AT & T email system [1]. They provide results showing the effectiveness of finding faults using all two-way interactions. D. Cohen *et al.* extend these ideas in the development of the *Automatic Efficient Test Generator (AETG)* [5]. This is a commercially available test case generator developed at Telcordia Technologies. Dalal *et al.* and Burr *et al.* show that two-way interaction coverage using AETG identifies a large number of software interaction faults and provides good code coverage [2, 11]. 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 [12]. 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[14]. Williams *et al.* suggest the use of this method for component and network interaction testing [23, 24].

We return to the example in Table 1. Table 2 shows a set of test cases for this system. Each row is a test in which each component has exactly one configuration selected. The first test case, (RAID 5, Novell, 128 MB, Ultra 160-SATA), covers six two-way interactions (RAID 5 with Novell, RAID 5 with 128 MB of memory, RAID 5 with an Ultra 160-SATA disk interface, Novell with 128 MB of memory, Novell with an Ultra 160-SATA interface, and 128 MB of memory with an Ultra 160-SATA interface) or four three-way interactions (RAID 5 and Novell with 128 MB, RAID 5 and Novell with Ultra 160-SATA, RAID 5 and 128 MB with Ultra 160-SATA, and Novell and 128 MB with Ultra 160-SATA). There are a total of $\binom{4}{2} \times 3^2$ or 54 two-way interactions to test. For the data in Table 1 we can test all of these interactions with nine test cases, or all of the 108 three-way interactions with the 27 test cases shown.

The test suite shown in Table 2 is equivalent to a combinatorial design called a *covering array*. A *covering array*, $CA(t, k, v)$, of *size* N is an $N \times k$ array such that every $N \times t$ sub-array contains all ordered subsets from v symbols of size t *at least* once. The notation $CA(N; t, k, v)$ is also used, in order to indicate the size explicitly. Table 2 is an example of a $CA(27; 3, 4, 3)$.

At the current time there are two distinct areas of active research on combinatorial designs for software testing. The mathematics community is focusing on building smaller covering arrays of higher interaction strength [4, 17, 19, 20]. The software testing community is focusing on greedy search algorithms to build these in a more flexible environment, one that more closely matches real testing needs [5, 6, 11, 12, 23, 22, 25]. Ideally we would like to combine these ideas and build higher strength interaction tests that are minimal and efficient to generate.

As the methods of building covering arrays for testing are varied, a trade off must occur between computational power and the cost of running the test suites. In some software testing applications, the removal of a small number of tests is not a primary goal. However, when tests are to be run repeatedly, or there is a large setup cost in executing the tests, it is more efficient to invest in test suite minimization and save on test suite execution costs. In this paper we examine some methods of combining both computational search and algebraic construction to efficiently build optimal test suites with this last idea in mind.

2. Combinatorial Objects

The problems faced in software interaction testing are not unique. Similar problems exist for testing in other disciplines such as agriculture, pharmaceuticals, manufacturing and medicine [13]. The primary combinatorial objects used to satisfy the coverage criteria for these types of problems are *orthogonal arrays* and *covering arrays*. We begin with a few definitions and then describe how these objects can be applied to software testing.

1	2	1	1
2	0	2	1
2	1	1	0
1	1	2	2
0	2	2	0
1	0	0	0
2	2	0	2
0	1	0	1
0	0	1	2

Table 3. An $OA(9; 2, 4, 3)$

RAID 1	Novell	128	Ultra 160-SCSI
RAID 5	XP	256	Ultra 160-SCSI
RAID 5	Linux	128	Ultra 320
RAID 1	Linux	256	Ultra 160-SATA
RAID 0	Novell	256	Ultra 320
RAID 1	XP	64	Ultra 320
RAID 5	Novell	64	Ultra 160-SATA
RAID 0	Linux	64	Ultra 160-SCSI
RAID 0	XP	128	Ultra 160-SATA

Table 4. Test suite derived from Table 3

2.1. Orthogonal Arrays

An orthogonal array $OA_\lambda(N; t, k, v)$ is an $N \times k$ array on v symbols such that every $N \times t$ sub-array contains all ordered subsets of size t from v symbols *exactly* λ times. Orthogonal arrays have the property that $\lambda = \frac{N}{v^t}$. When $\lambda = 1$ we can leave it out of the notation, and write $OA(N; t, k, v)$. An $OA(N; t, k, v)$ is a special type of $CA(N; t, k, v)$. Table 3 is an example of an $OA(9; 2, 4, 3)$. If we select any two columns from this array it has the property that each possible ordered pair from the symbols $0, 1, 2$ occurs exactly one time. This array uses the symbols $0, 1, 2$ in all of the columns, but since the properties we are interested in occur only *between* columns, the meaning of each of these symbols in each column is different. We can remap the symbols for each column arbitrarily without losing the desired properties. For instance we can map 0 in the first column to RAID 0, and 0 in the second column to Windows XP. If we map each symbol from each column to one configuration of a component we have transformed this orthogonal array into a test suite. Table 4 is a test suite for finding all 2 way interactions from Table 1 which is derived from the orthogonal array shown in Table 3.

We do not need such a stringent object for software testing. In fact orthogonal arrays may be too restrictive as they only exist for certain values of t, k, v. Instead we can use a *covering array* that allows some duplication of coverage.

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

Table 5. $CA(5; 2, 4, 2)$

2.2. Covering Arrays

In a covering array $CA(N; t, k, v)$, t is called the interaction *strength*, k the *degree*, v the *order*, and N the *size*.

Table 5 gives an example of a $CA(5;2,4,2)$. Adding one row to this covering array yields a $CA(6;2,4,2)$. Since covering arrays can have arbitrarily many rows, one aim is to use the smallest possible number of rows to satisfy the properties of a covering array. The *covering array number*, $CAN(t,k,v)$, is the minimum number N for which a $CA(N;t,k,v)$ exists. For example, $CAN(2,5,3) = 11$ [4]. Many covering array numbers are unknown, therefore the literature usually reports the upper and lower bounds for particular array sizes. The lower bound is the smallest theoretical size for an array, while the upper bound is the smallest size for which an array is known to exist. We are interested in upper bounds for our problem since we need to actually build these.

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 $t = 3$,$k = 4$, $v = 3$, and $N = 27$. Each component is represented by one column; each row is an individual test of the test suite.

In software systems, of course, the numbers of configurations for each component can vary in size. We define a more general object to describe this variability (see [8] for a more in-depth discussion). A *mixed level* covering array, $MCA(N;t,k,(v_1,v_2,...,v_k))$, is an $N \times k$ array on v symbols, where $v = \sum_{i=1}^{k} v_i$, with the following properties:

1. Each column $i \quad (1 \leq i \leq k)$ 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. Constructing Covering Arrays for Test Suites

Most of the literature pertaining to covering arrays for software testing involves methods for building these in an efficient manner while obtaining small test suites. The mathematical literature primarily presents new algebraic constructions [4, 6, 17, 19, 20], while the software engineering literature proposes new greedy algorithms [5, 22, 25], as well as standard heuristic search techniques such as simulated annealing [7, 8]. AETG uses a greedy algorithm to find test cases. It handles an arbitrary strength t as well as special conditions. This includes seeding specific test cases and avoiding particular interactions. Two other greedy algorithms, In Parameter Order (IPO), and Test Case Generator (TCG), focus only on the case when $t = 2$ [22, 25].

When $t = 3$, the combinatorial research illustrates both the depth of the connection with combinatorial configurations and the difficulties that these pose for software testers. The techniques applied to date when $t = 3$, at least in the range of *small* covering arrays, vary greatly. They range from very simple construction methods such as identifying distinct symbols to form a single symbol, through to more complex *cut-and-paste* constructions. These are constructions that use smaller covering arrays as building blocks. Finally sophisticated recursive constructions exist that combine small covering arrays but also employ related combinatorial objects. While the more sophisticated constructions yield substantially smaller covering arrays when they can be applied, these same constructions do not apply as generally as we require. (For a summary of known results when $t = 3$ see [4].)

We do, however, need to be able to apply stronger interaction testing such as 3-way testing to software test suites [12, 14]. In addition, Cohen *et al.* have suggested the need for focusing stronger interaction testing on subsets of components where faults are likely to occur or be too costly [7, 8]. In the rest of this paper, therefore, we examine a small group of constructions for the case of $t = 3$ using augmented cut-and-paste techniques.

The true challenge facing the software tester is to determine when the construction applies, including what auxiliary ingredients are needed. This overhead limits the applicability of the more complex constructions. We do not attempt to solve that problem here but acknowledge that this is an open and interesting problem.

3.1. Computational Search

Computational search techniques to find covering arrays include greedy algorithms and standard combinatorial search techniques such as simulated annealing [5, 8, 22, 25]. We use simulated annealing, a search technique for solving combinatorial optimization problems, that has shown to have good general results for finding minimal test suites especially when the problem size is relatively small [8]. In [8] simulated annealing was compared with known greedy algorithms. When test suite minimization is the goal, this algorithm works better than the greedy methods. Some other heuristic search techniques have been tried in [18] for the case of $t = 2$, but were found to be less effective than annealing [18]. Our annealing program is patterned on that of [16].

3.2. Simulated Annealing

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

To start, we randomly choose an initial feasible solution. In our problem this corresponds to an $N \times k$ array with symbols randomly chosen from the desired covering array specification. The algorithm then generates a sequence of trials, in which we randomly select a cell of this array and change the symbol to a new one. If the transition results in a feasible solution S' of lower or equal cost then S' is accepted. If it results in a feasible solution of higher cost, then S' is accepted with probability $e^{-(c(S')-c(S))/T}$, where T is the controlling temperature of the simulation. The cost in our problem is the number of uncovered t-sets. A cost of zero indicates we have a covering array. The temperature is lowered in small steps with the system being allowed to approach “equilibrium” at each temperature through a sequence of trials at this temperature. Usually this is done by setting $T := \alpha T$, where α (the *control decrement*) is a real number slightly less than one. After an appropriate stopping condition is met, the current feasible solution is taken as the solution of the problem at hand. The idea of allowing a move to a worse solution helps to keep the solution from being stuck in a bad state, while continuing to make progress. The algorithm stops once a feasible solution of cost zero is obtained or we are frozen.

Since N is unknown at the start of our search we select an arbitrarily large N and repeat the annealing process multiple times using a binary search technique, keeping the smallest N x k array that finishes without freezing. For each of the arrays found in this paper we ran the annealing program three to five times and selected the array with the smallest N. We used similar temperature and cooling schedules each time which gave us a small variance in the overall size of N for each array. We find that a starting temperature of approximately .20 and an α between 0.9998 and 0.99999 every 2000 iterations works well for this size problem.

Based on [8] it appears that simulated annealing does well when the search space is small and there are abundant solutions. As the search space increases and the density of potential solutions becomes sparser the algorithm may fail to find a good solution or may require extremely long run times. Careful tuning of the parameters of temperature and cooling can improve upon the results, but at a potential computational cost. Cohen *et al.* present results suggesting that annealing works well for covering arrays, often produces smaller test suites than other computational methods, and sometimes improves upon algebraic constructions, but it fails to match the algebraic constructions for larger problems, especially when $t = 3$ [8].

3.3. Algebraic Constructions

Algebraic constructions often provide a better bound in less computational time than heuristic search. However, they are not as general and must be tailored to the problem at hand. An in-depth knowledge base must exist to decide which construction best suits a particular problem. In addition, most constructions aim at proving the existence of a class of objects. In software testing we may not always need the absolute minimum sized test suite, but want to get consistently good results that are close to minimum. We also would like to obtain these results in a computationally feasible amount of time.

4. Combining Methods

The idea of using small building blocks to construct a larger array is used often in algebraic constructions. We refer to these techniques in general as *cut-and-paste* methods. Is it possible to use a combinatorial construction and augment this with heuristic search to allow one to ‘construct’ an array with limited understanding of the underlying combinatorics? The rest of this paper examines how one can use such a combined approach. It presents some initial results suggesting that this technique performs better than heuristic search alone, while it is more flexible than a straight algebraic construction.

4.1. Construction Using an Ordered Design

Cohen *et al*. [8] found a $CA(300; 3, 6, 6)$ using simulated annealing. This is smaller than the previously reported $CA(305; 3, 6, 6)$, found using an algebraic construction [4]. It is one of the few cases where simulated annealing alone, improves upon an algebraic construction. Unfortunately the annealing process for this array takes considerable computational time. In addition, as the problems grow larger, the less likely annealing is to find a solution close to that of an algebraic construction. Since there is a trade-off between the time it takes to build a test suite and the time it takes to set up and run it, we are interested in methods that reduce the number of tests (i.e. find smaller sized covering arrays), and are also quicker to create.

One can improve upon both of these reported upper bounds for $CA(3, 6, 6)$ using a construction that divides

3	1	2	0
1	2	3	0
2	3	1	0
3	2	1	0
1	3	2	0
2	1	3	0
0	1	2	3
0	2	1	3
1	2	0	3
2	1	0	3
2	0	1	3
1	0	2	3
0	2	3	1
0	3	2	1
3	2	0	1
2	3	0	1
3	0	2	1
2	0	3	1
0	3	1	2
0	1	3	2
3	1	0	2
1	3	0	2
3	0	1	2
1	0	3	2

Table 6. Ordered Design: $OD(3, 4, 4)$

the problem into smaller pieces. Additionally, since we only need to search for smaller objects the time to build this array is also reduced. For instance we shall obtain a $CA(263; 3, 6, 6)$ and we shall create a $CA(1224; 3, 10, 10)$ which is smaller than the 1331 rows reported in [4] or that of using simulated annealing alone (2163). We develop a general strategy, using the following result to provide an example of the constructive technique.

We begin by defining another combinatorial object. An *ordered design*, $OD(t, k, v)$, is a $\binom{v}{t} \cdot t! \times k$ array in which each row has k distinct entries and every t columns contains every row tuple of t distinct entries exactly once. Table 6 is an example of an $OD(3, 4, 4)$. In this example all ordered tuples of size three from the set $\{0, 1, 2, 3\}$ in which the three entries are different occurs exactly once in any three columns. For instance, the tuples $\{(0, 1, 2), (0, 1, 3), (0, 2, 3), (0, 2, 1), (0, 3, 1), (0, 3, 2)\}$ appear in each. But the tuple $(0, 0, 1)$ never occurs.

Theorem 1 *An ordered design of the form $OD(3, q + 1, q + 1)$ exists when q is a prime power [9].*

To build the $CA(3, 6, 6)$ we start with an $OD(3, 6, 6)$. This has 120 rows. By definition it contains all three-way interactions between columns that are of the form (a, b, c) where $a \neq b \neq c \neq a$. To satisfy the conditions of a covering array, all combinations of triples containing only two unique symbols, i.e. $(a, a, b), (a, b, a), (a, b, b)$, etc., must also occur. We handle that case now. We first create a $CA(12; 3, 6, 2)$. We can leverage the fact that it uses only two symbols. There are $\binom{6}{2} = 15$ pairs of symbols

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

Table 7. $CA(12; 3, 6, 2)$ **with 2 disjoint rows**

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

Table 8. $CA(13; 3, 10, 2)$ **with 2 disjoint rows**

in this array. If we make 15 copies of this covering array and map symbols for each so that all 15 combinations of pairs have been used we have all of the possible triples between columns containing two unique symbols. We append these arrays to our ordered design which gives us $120 + (15 \times 12) = 300$ rows. This is a $CA(300; 3, 6, 6)$.

A simple improvement can be made. Without changing the structure of the covering array $CA(3, 6, 2)$, we can relabel the symbols in the array so that the array always contains a test of the form a, a, a, a, a, a where a is the symbol in the array with the smallest cardinality. If we then apply the above construction, the row 0,0,0,0,0,0 appears five times and in general the row i, i, i, i, i, i appears at least $5 - i$ times. Consequently, ten duplicate rows can be removed to establish that $CAN(3, 6, 6) \leq 290$. An alternate method, which improves even further upon the original size of 300, is to use $CA(3, 6, 2)$s containing two disjoint rows. Table 7 is such an array. Any two disjoint rows in such a covering array can be remapped to the form a, a, a, a, a, a and b, b, b, b, b, b. Since all six symbols are covered five times we can remove the 30 rows of this form leaving us with 270. We must add back one row of the form a, a, a, a, a, a for each symbol which establishes that $CAN(3, 6, 6) \leq 276$.

We can apply this same method to build a $CA(1225; 3, 10, 10)$. We begin with an $OD(3, 10, 10)$ of size 720. In the same manner as above we can append

$+$	0	1	2	3	4	5	6	6	8
0	0	1	2	3	4	5	6	7	8
1	1	2	0	4	5	3	7	8	6
2	2	0	1	5	3	4	8	6	7
3	3	4	5	6	7	8	0	1	2
4	4	5	3	7	8	6	1	2	0
5	5	3	4	8	6	7	2	0	1
6	6	7	8	0	1	2	3	4	5
7	7	8	6	1	2	0	4	5	3
8	8	6	7	2	0	1	5	3	4

Table 9. GF(9) addition table

$\times$	0	1	2	3	4	5	6	6	8
0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8
2	0	2	1	6	8	7	3	5	4
3	0	3	6	7	1	4	5	8	2
4	0	4	8	1	5	6	2	3	7
5	0	5	7	4	6	2	8	1	3
6	0	6	3	5	2	8	7	4	1
7	0	7	5	8	3	1	4	2	6
8	0	8	4	2	7	3	1	6	5

Table 10. GF(9) multiplication table

45 $CA(13;3,10,2)$s including two disjoint blocks (see Table 8). This gives us 1305 rows. We now remove the 90 disjoint blocks and add back in 10 blocks of the form $(a, a, ..., a)$, one for each symbol. This gives us a total of 1225 rows.

4.2. OD Construction

There are two problems with this construction. The first is that we must construct the ordered design, (i.e. the existence is not enough) if we want to use this in a practical environment. We outline a method here to do this which has been adapted from [3]. Let q be a prime or power of a prime. We choose a set Q of q elements, which contains 0 and 1, and $q-2$ other symbols. We add a $(q+1)$st symbol ∞ to this set. Now we consider certain permutations defined on $Q \cup \{\infty\}$ as follows. Choose in all possible ways 4-tuples of values (a, b, c, d) with a, b, c, d in Q subject to the conditions that

1. $a = 1$, c arbitrary, b, d arbitrary but d cannot be a multiple of $b \times c$; and
2. $a = 0$, $c = 1$, b, d arbitrary but b not equal to 0.

For each such selection (a, b, c, d), we make a permutation (test case) in the following way. We use symbols $0...|Q|-1$

0	1	2	3	4	5	6	7	8
NA	1	2	4	3	7	8	5	6

Table 11. GF(9) inverses

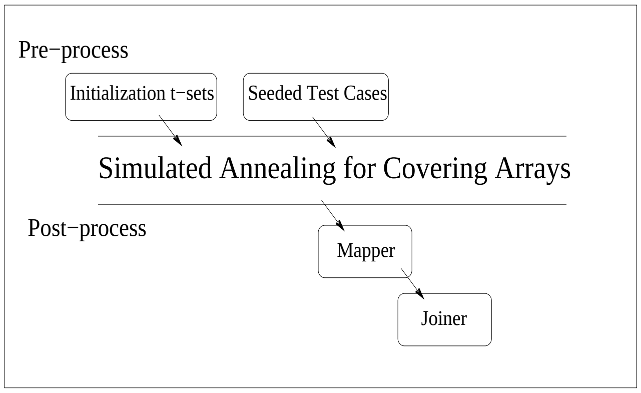

Figure 1. Augmented annealing process

to represent the elements of Q. Now $Q \cup \{\infty\}$ indexes the factors (or columns). For factor x, we set the value to a/b if $x = \infty$, and $b \neq 0$, to ∞ if $x = \infty$ and $b = 0$, to ∞ if $bx + d = 0$, and to $(ax + c)/(bx + d)$ otherwise.

In order to do the arithmetic for both of these steps, the set Q needs some structure. The easiest case is when q is a prime. Then Q is the set of integers modulo q. Multiplying and adding are the same as usual but the result is reduced modulo q. To divide we just multiply by the inverse. For example, a/b is a $\times$ b^{-1}, where b^{-1} is the number which when multiplied by b gives 1. We can use the Extended Euclidean Algorithm to find inverses modulo q when q is prime [21].

When q is a prime power, we must use a finite field to multiply, add and find inverses. We don't attempt to describe the construction of a finite field here, but provide the addition, multiplication and inverse tables used for $q = 9$ in Tables 9-11. We can use any of a number of algebraic computer packages to produce a finite field, or alternatively, we can create one by hand (see [21]).

The second problem is that we must be able to determine the existence of two covering arrays with disjoint rows and then construct these. There is no general proof of the existence of these arrays. We have instead used simulated annealing to construct these and present them in Tables 7-8.

5. Augmented Annealing

In the previous section we described a construction using an ordered design, which can produce smaller covering arrays than the best known algebraic constructions. There are two problems that have been mentioned, and that may be handled differently by a software tester and a mathematician. The software tester must actually construct these arrays, not just prove they exist. In this case, the construction to create an ordered design requires a separate algorithm and may require the use of finite field arithmetic.

We present a method below to avoid this construction. For some of the smaller cases such as $CA(3, 6, 6)$ this method works very well. On larger problems using the actual construction certainly is preferred if the test suite size is of importance, but if the only toolkit the tester has is simulated annealing then this method is one which can be employed.

The second problem, which is imperative for proof of existence is probably less worrisome for the software tester. In this case, proving that we have a $CA(3, 6, 2)$ with two disjoint rows can be done using heuristic search. If the search produces an array of slightly larger size we can nonetheless get a nearly optimal test suite that appears to be smaller than the one built from straight annealing. In this section we address how to augment the simulated annealing program to handle these two problems. In addition we provide some constructions using the augmented annealing algorithm to improve further upon the bounds given above.

We have augmented the simulated annealing program with several modules as is shown in Figure 1. Our aim is to use our annealing program to construct small building blocks which can be joined together. We have added two methods for building *partial* covering arrays. The first method is an initialization method. The second method adds a set of fixed (or seeded) test cases. We have also added some post processing methods. The first is a mapper and the second is a joiner.

5.1. Modules

The initialization method reads in a subset of t-sets and counts these as covered. The annealing proceeds to build a potentially incomplete covering array since it believes these initialization t-sets are not needed. Therefore a move to a feasible solution that adds one of these t-sets will not improve our solution and is rarely chosen. We do not explicitly exclude these from being covered, but see this as a potential further enhancement. We can use this to build an ordered design of a small size, by initializing it with all triples which have repeated symbols, i.e. (a, a, b) and to build partial arrays that cover all triples excluding those of type (a, b, c) found in the ordered design. In our experimentation we have found that the second problem is easier for annealing than the first where fewer solutions exist in the search space. We believe this can be used in other cut-and-paste constructions allowing us to build individual partial arrays of other types.

The AETG system includes the ability to add seeded test cases to a test suite [5]. These are test cases that the tester wants to run each time, regardless of coverage. In any real test situation, one should have the ability to choose a set of tests that must be run. We have included the ability to add seeded test cases to our program. It counts these as part of the covering array to be built, but it does not alter them. The seeds are *fixed*, i.e. no changes can be made to their values during annealing. We can use seeded test cases that span entire rows of the array or partial rows of the array. In this case the non-fixed part of the test cases can be changed. The covered t-sets are counted, but the program must do all of its annealing excluding these positions. We use this module, for example, to seed the arrays with disjoint rows.

The symbol mapping for a covering array is arbitrary and can be remapped as long as we use v unique symbols for each column of the covering array. We see this in Tables 3 and 4. When we build the smaller covering arrays with disjoint rows, we may want to use the same array repeatedly, with different symbol mappings. Therefore, a mapper is used to translate arrays from one symbol set to another.

Lastly, when building test suites using cut and paste techniques such as are presented here we end up with pieces that must be merged together. A module that joins test suites together both horizontally and vertically has been added for this purpose.

6. Results

Given the augmented annealing program one can follow the construction described above and create minimal test suites. This can be improved upon with some slight variations. Algebraic constructions often force more coverage than is really necessary in order to simplify the proof of the existence of the object being constructed. In building arrays for software testing, we want arrays that are as small as possible, but are not necessarily interested in proving new bounds in a general manner.

6.1. Constructions for CA(3,6,6)

We have applied the ordered design technique above to some covering arrays of the form $CA(3, q+1, q+1)$ as well as to a covering array which does not have an ordered design and a mixed level array. In [8] the upper bound reported for $CAN(3, 6, 6)$ using straight simulated annealing was 300. We have used the above construction in conjunction with the augmented annealing to establish $CAN(3, 6, 6) \leq 263$. This construction is described at the end of this section.

We return to the construction presented in Section 4.1. We create the ordered design by annealing a partial covering array using the initialization module of the augmented program. It is initialized with all triples not of the type (a, b, c). The known bound of 120 is easily obtained. Next a $CA(12; 3, 6, 2)$ with two seeded rows of the type $(a, a, .., a)$ and $(b, b, ..., b)$ is created. The two disjoint rows are removed and this array is passed to the mapper. It creates 15 arrays of size 10, containing all of the possible symbol combinations of type (a, a, b), $(a, b, b)(a, b, a)$, (b, a, b), etc. The six rows of type $(a, a, ..., a)$ are created and then all of the pieces are joined. The final covering array is of size 276.

We can create variations on this construction since we are not necessarily restricted to this combination of elements. An ordered design covers all triples containing three unique symbols, so we are only concerned with covering all combinations of pairs of symbols from $CA(3, 6, 6)$.

We have constructed a partial $CA(3, 6, 3)$ initialized with all of the triples of the type (a, b, c). We can get a partial covering array of size $N = 30$. The bound for a complete covering array of this size is 33 so we have saved some rows by using the initialization method. We can use this to cover $\binom{3}{2} = 3$ combinations of the six pairs of symbols. There are still 12 remaining. We can cover these using 12 $CA(12; 3, 6, 2)$s with two disjoint rows a, a, a, a, a, a and b, b, b, b, b, b removed. Each of these now contains ten rows. Lastly we add back in three rows of type a, a, a, a, a, a (we can exclude the three symbols covered by the $CA(3, 6, 3)$) and join these together. This gives us a covering array of size $120 + 30 + (12 \times 10) + 3 = 273$. This is smaller than the first construction above. Of course all of the other combinations of $CA(3, 6, i)$, $i \leq 6$, that cover all pairs of symbol combinations can be tried in this manner. For instance we can also use a $CA(3, 6, 4)$ combined with nine $CA(3, 6, 2)$s. Exploration of various combinations is needed to determine the smallest array size. The final array sizes using different combinations may vary. Ideally we see the need to pre-compute various combinations of arrays so we can choose the best decomposition.

In the case of $CA(3, 6, 6)$ we found the smallest array using only two building blocks. We used annealing to create an ordered design of size 120 and annealing to create a partial $CA(3, 6, 6)$ of size 143, initialized with all triples of type (a, b, c). This gives us a $CA(263; 3, 6, 6)$. It improves upon the other constructions and provides us with a new bound.

These alternate constructions can be used in all of the cases outlined below, but we have restricted the discussion from now on to two methods. The first method (A) uses an ordered design and a partial $CA(3, k, v)$ initialized with all triples of type (a, b, c). The second method (B), creates an ordered design and $\binom{k}{2}$ $CA(3, k, 2)$s each with two disjoint blocks removed, plus k rows of type $a, a, a, ..., a$. These two constructions are illustrated in Figure 2.

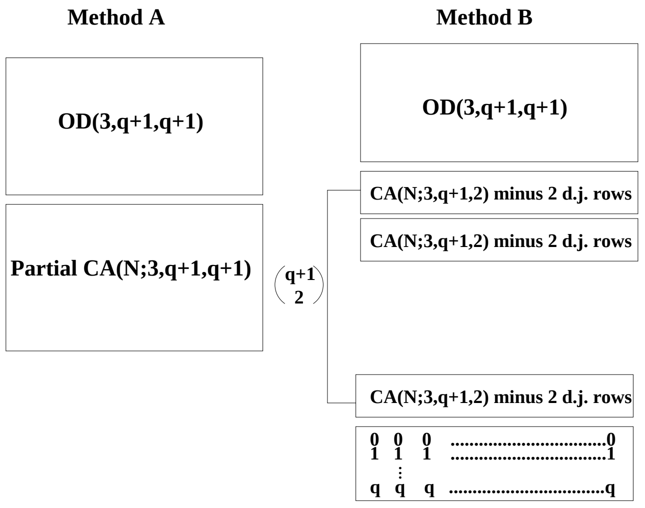

Figure 2. Constructions using an ordered design

6.2. Constructions for CA(3,8,8)

In the case of $CA(3, 8, 8)$ and $CA(3, 9, 9)$, it is known that the collection of all triples can be covered exactly, i.e. every triple is covered precisely once (this is an orthogonal array of strength three which is optimal). We therefore do not expect any improvement over the best known result using our method. However, the smallest array we have managed to find using simulated annealing in a reasonable amount of computational time for the $CA(3, 8, 8)$ has 918 rows. This is considerably larger than the known orthogonal array size of 512 and required almost three hours to run. We can instead create a $CA(616; 3, 8, 8)$ in significantly less computational time. We use an $OD(3, 8, 8)$ of size 336 created with the direct construction given in Section 4.2 and anneal a partial covering array of size 280 in approximately five minutes. This provides us with a $CA(616; 3, 8, 8)$ which is smaller and computationally less expensive than using only annealing.

6.3. Constructions for CA(3,10,10)

For the $CA(3, 10, 10)$ we can use the ordered design construction from Section 4.2 to generate the first part of this array. We can build 45 $CA(13; 3, 10, 2)$s minus the two disjoint blocks and add back in 10 rows of type $a, a, .., a$. If we do this we have an array of size 1225 which improves upon the published bound of 1331 [4]. We can also build a partial $CA(3, 10, 10)$ using annealing. This gives us a partial array of size 504. When combined with the ordered design this in a test suite of size 1224. In comparison, the smallest array we have built with straight annealing for a $CA(3, 10, 10)$ is of size 2163.

Table 12 shows the smallest covering arrays found using the two augmented methods and provides the smallest numbers we have obtained using straight annealing as well as known bounds published in [4, 8]. In each case we ran our program several times but report only the smallest ar-

$CA(t,k,v)$	Augmented Annealing A	Augmented Annealing B	Simulated Annealing	Smallest Reported[1] Array Size
$CA(3,6,6)$	**263**	276	300	300
$CA(3,8,8)$	616	624	918	512
$CA(3,9,9)$	940	909	1490	729
$CA(3,10,10)$	**1224**	1225	2163	1331
$CA(3,12,12)$	2339	**2190**	4422	2197
$CA(3,14,14)$	4134	**3654**	8092	4096

Table 12. Sizes for covering arrays using augmented annealing.

Method A = OD + partial array, Method B = OD + $\binom{k}{2}CA(3,k,2)$s
1. Source = Chateauneuf *et al.*[4] and Cohen *et al.* [8]

Partial CA	Size	Ordered Design
$CA(3,6,6)$	143	120
$CA(3,8,8)$	280	336
$CA(3,9,9)$	436	504
$CA(3,10,10)$	504	720
$CA(3,12,12)$	1019	1320
$CA(3,14,14)$	1950	2184

Table 13. Sizes for partial CAs and ordered designs

ray found. The variation in results was small, but due to the randomness in annealing we do not always produce the same size array each time. The first method, labeled A, uses an ordered design and anneals a partial array initialized with the triples covered in the ordered design. The second method uses an ordered design and combines it with $\binom{k}{2}$ $CA(3,k,2)$s each with 2 disjoint rows removed and one row added back for each of the k symbols. The best values we have found for arrays with disjoint rows are given in Table 14. The ordered design for $CA(3,6,6)$ was created using annealing. All of the other ordered designs were created with a program that implements the construction given in section 4.2. Values in bold font are new upper bounds for these arrays.

CAs w/2 disjoint rows	Size
$CA(3,6,2)$	12
$CA(3,8,2)$	12
$CA(3,9,2)$	13
$CA(3,10,2)$	13
$CA(3,12,2)$	15
$CA(3,14,2)$	18

Table 14. Sizes for covering arrays with 2 disjoint rows

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

Table 15. Sizes for covering arrays with no known algebraic constructions

6.4. Arrays with No Known Algebraic Constructions

We end with some examples which do not have ordered designs as part of their makeup to illustrate that we can still these techniques when the problems are not as simple as those given so far. Real software test environments often do not have parameters that match known constructions and can have different numbers of configurations per component, i.e. they are mixed level.

The first example is a $CA(3,7,7)$. We have tried using annealing to create partial arrays with and without the triples of type (a,b,c) as if an ordered design exists. We only improve very slightly on the best bound found for this array from straight annealing, but believe that we improve on the computational time that is required to solve this problem. The second example is an $MCA(3,6^6 4^2 2^2)$. This array contains a $CA(3,6,6)$ but has four additional components. We have tried several techniques to build this array. When straight annealing was used we found an array of size 317, which is much larger than the best bound we have found for the sub-array $CA(3,6,6)$. Based on [7] we believe that the hardest problem, that of the $CA(3,6,6)$ drives the final size of this array so we have tried other variations. When we used two partial covering arrays as in Method A, the best bound we found was 313. We tried instead to seed this array with the harder problem already solved. We use the seeding module to build the array by seeding and fixing either the $OD(3,6,6)$ of size 120 or the $CA(3,6,6)$ of size 263 and anneal to add the additional structure. Both of these improve markedly upon the first two methods as shown in Table 15. The smallest test suite we found was by using the $CA(3,6,6)$ as a seed. We added only 9 test cases to complete the missing coverage. Of course this highlights the need for the software tester to have some sort of knowledge base to determine which method is best for which problem.

7. Conclusions

We have presented a method for combining algebraic constructions and simulated annealing to build covering arrays of strength three for software testing. These methods are most interesting in testing situations where the setup costs are large and the goal is to test as many interactions with as few tests as is possible. We have focused on strength three covering arrays. These are harder to build using simple heuristic search techniques due to the explosion in the search space. There are many known algebraic constructions, however these are sometimes too restrictive or require a priori knowledge about additional constructions that the software tester may lack. For instance we have given an example construction where an *ordered design* is required. In order for statistical design of experiment methods to be useful in software testing these issues must be addressed.

This paper presents one way to bridge the gap between algebraic and computational methods. One can relax the proof requirements found in algebraic constructions for use in a practical environment when it is not of interest to prove classes of objects exist, but to build objects that are as small as possible in a reasonable computational time. By augmenting a simulated annealing algorithm we can build partial covering arrays to satisfy certain conditions and then remap and combine these to build minimal test suites. We have improved upon a few reported upper bounds for strength three covering arrays. The randomness in annealing, though, does force us to use this in cases where an approximation to a best solution is needed since we cannot guarantee a final bound.

These methods are not entirely useful for interaction testing on their own. The tester still requires knowledge of which constructions and methods are best for appropriate problem sizes. Producing such a toolkit is an open and interesting problem. In order to attempt this, a knowledge base containing a core set of construction methods and the best fit for particular problems is required. In this paper we have examined one more technique which can ultimately be added to this set.

We have used only a small set of fixed level arrays as examples for these constructions. We believe that these techniques can be extended and used for a larger variety of mixed level arrays and arrays for which particular constructions do not exist. We also see the need for an empirical study to assess the ease and effectiveness of our methods.

Acknowledgments

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

References

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

[2] 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.

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

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

[5] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–44, 1997.

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

[7] M. B. Cohen, C. J. Colbourn, J.S. Collofello, P. B. Gibbons and W. B. Mugridge. Variable Strength Interaction Testing of Components. In *Proc. of the Intl. Computer Software and Applications Conference, (COMPSAC 2003)*,2003, Dallas TX, to appear.

[8] 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 OR.

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

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

[11] 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.

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

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

[14] 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.

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

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

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

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

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

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

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

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

[23] 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.

[24] 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.

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