Variable Strength Interaction Testing of Components

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

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

Abstract

Complete interaction testing of components is too costly in all but the smallest systems. Yet component interactions are likely to cause unexpected faults. Recently, design of experiment techniques have been applied to software testing to guarantee a minimum coverage of all t-way interactions across components. However, t is always fixed. This paper examines the need to vary the size of t in an individual test suite and defines a new object, the variable strength covering array, that has this property. We present some computational methods to find variable strength arrays and provide initial bounds for a group of these objects.

1. Introduction

In order to shorten development times, reduce costs and improve quality, many organizations are developing software utilizing existing components. These may be commercial off-the-shelf (COTS) or internally developed for reuse among products. The utilization of existing components requires new development and verification processes [2]. In particular the interaction of these new components with each other as well as with the newly developed components within the application must be tested.

Software is becoming increasingly complex in terms of components and their interactions. Traditional methods of testing are useful when searching for errors caused by unmatched requirements. However, component based development creates additional challenges for integration testing. The problem space grows rapidly when searching for unexpected interactions. Suppose we have five components, each with four possible configurations. We have $4^5 = 1,024$ potential interactions. If we combine 100 components there are 4^{100} possible combinations. This makes testing all combinations of interactions infeasible in all but the smallest of systems.

Instead one can create test suites that guarantee pairwise or t-wise coverage. For instance we can cover all pairwise interactions for ten components, each with four possible configurations, using only 25 test cases. A *covering array*, $CA(N; t, k, v)$ is an $N \times k$ array such that every $N \times t$ subarray contains all ordered subsets from v symbols of size t at least once. The *covering array number* is the minimum N required to satisfy the parameters t, k, v.

Covering arrays have been used for software interaction testing by D. Cohen *et al.* in the *Automatic Efficient Test Generator* (AETG) [5]. Williams *et al.* use these to design tests for the interactions of nodes in a network [12]. Dalal *et al.* present empirical results suggesting that testing of all pairwise interactions in a software system indeed finds a large percentage of existing faults [7]. In further work, Burr *et al.* provide more empirical results to show that this type of test coverage is effective [3].

In a software test, the columns of the covering array represent the k *components* or fields. Each component has v *levels* or configurations. The final test suite is an $N \times k$ array where N is the number of test cases and each test contains one configuration from each component. By mapping a software test problem to a covering array of strength t we can guarantee that we have tested all t-way interactions.

In many situations pairwise coverage is sufficient for testing. However, we must balance the need for stronger interaction testing with the cost of running tests. For instance a $CA(N; 2, 5, 4)$ can be achieved for as little as 16 tests, while a $CA(N; 3, 5, 4)$ requires at least 64 tests. In order to appropriately use our resources we want to focus our testing where it has the most potential value.

The recognition that all software does not need to be tested equally is captured in the concept of risk-based testing [1]. Risk-based testing prioritizes testing based on the probability of a failure occurring and the consequences should the failure occur. High risk areas of the software are identified and targeted for more comprehensive testing.

The following scenarios point to the need for a more flexible way of examining interaction coverage.

- We completely test a system, and find a number of components with pairwise interaction faults. We believe this may be caused by a bad interaction at a higher strength, i.e. some triples or quadruples of a group of components. We may want to revise our testing to handle the “observed bad components” at a higher strength.
- We thoroughly test another system but have now revised some parts of it. We want to test the whole system with a focus on the components involved in the changes. We use higher strength testing on certain components without ignoring the rest.
- We have computed software complexity metrics on some code, and find that certain components are more complex. These warrant more comprehensive testing.
- We have certain components that come from automatic code generators and have been more/less thoroughly tested than the human generated code.
- One part of a project has been outsourced and needs more complete testing.
- Some of our components are more expensive to test or to change between configurations. We still want to test for interactions, but cannot afford to test more than pairwise interactions for this group of components.

While the goal of testing is to cover as many component interactions as possible, trade-offs must occur. This paper examines one method for handling *variable interaction strengths* while still providing a base level of coverage. We define the variable strength covering array, provide some initial bounds for these objects and outline a computational method for creating them.

2. Background

Suppose we are testing new integrated RAID controller software. We have four components, (RAID level, operating system (OS), memory configuration and disk interface). Each one of these components has three possible configurations. We need 81 tests to test all interactions. Instead we can test all pairwise interactions of these components with only nine tests. Perhaps though, we know that there are more likely to be interaction problems between three components: RAID level, OS and memory. We want to test these interactions more thoroughly. But it may be too expensive to run tests involving all three way interactions among components. In this instance we can use three-way interaction testing among the first three components while maintaining two-way coverage for the rest. We still have a minimal coverage guarantee across the components and we still do not need to run 81 tests. The test suite shown in Table 2 provides this level of variable strength coverage with 27 tests.

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 6.x	256 MB	Ultra-160 SATA

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

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

Table 2. Variable strength array for Table 1

Commercial software test generators, like AETG, provide only a fixed level of interaction strength [5]. We might use this to build two separate test suites and run each independently, but this is a more expensive operation and does not really satisfy the desired criteria. We could instead just default to the higher strength coverage with more tests. However, the ability to tune a test suite for specific levels of coverage is highly desirable, especially as the number of components and levels increases. Therefore it is useful to define and create test suites with flexible strengths of interaction coverage and to examine some methods for building these.

3. Definitions

In a *covering array*, $CA(N;t,k,v)$, t is called the *strength*, k the *degree* and v the *order*. A covering array is optimal if it contains the minimum possible number of rows. We call this minimum number the *covering array number*, $CAN(t,k,v)$. For example, $CAN(2,5,3) = 11$ [4].

A *mixed level* covering array, denoted as a $MCA(N;t,k,(v_1,v_2,...,v_k))$, 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 can use a shorthand notation to describe our mixed covering array by combining v_i's that are the same. For example if we have three v_i's of size two we can write this 2^3. Consider an $MCA(N;t,(w_1^{k_1}w_2^{k_2}...w_s^{k_s}))$. This can also be written as an $MCA(N;t,k,(v_1,v_2,...,v_k))$ where $k = \sum_{i=1}^{s} k_i$ and $v = \sum_{i=1}^{s} k_i w_i = \sum_{i=1}^{k} v_i$. The following holds:

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

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

A *variable strength covering array*, denoted as a $VCA(N;t,(v_1,v_2,..,v_K),C)$, is an $N \times K$ mixed level covering array, of strength t containing C, a multi-set of disjoint mixed level covering arrays each of strength $\geq t$. We can abbreviate duplicate MCA's in the multi-set using a similar notation to that of the covering arrays. Suppose we have two identical covering arrays $CA(N;3,3^2)$ in C. This can be written as $CA(N;3,3^2)^2$. Ordering of the columns in the representation of a VCA is important since the columns of the covering arrays in C are listed consecutively from left to right.

An example of a $VCA(27;2,3^9 2^2,\{CA(27;3,3^3)^3\})$ can be seen in Table 3. The overall array is a mixed level array of strength two with nine columns containing three symbols and two columns containing two. There are three sub-arrays each with strength three. All three way interactions therefore among columns 0-2, 3-5, 6-8 are included. All two way interactions among all columns are also covered. This has been achieved with 27 rows which is the optimal size for a $CA(N;3,3^3)$. A covering array that would cover all three way interactions for all 11 columns, on the other hand, might need as many as 52 rows.

4. Construction Methods

Fixed strength covering arrays can be built using algebraic constructions if we have certain parameter combinations of t, k and v [4, 12]. Alternately we may use computational search algorithms [5, 6, 11]. Greedy algorithms form the basis of the AETG and the IPO generators [5, 11]. It has also been shown that simulated annealing is effective for building covering arrays of both mixed and fixed levels [6, 10]. Since there are no known constructions for variable strength arrays at the current time we have chosen to use a computational search technique to build these. We have written a simulated annealing program that has been used to produce all of the results presented in this paper.

4.1. Simulated Annealing

Simulated annealing is a variant of the state space search technique for solving combinatorial optimization problems. The hope is that the algorithm finds close to an optimal solution. Most of the time, however, we do not know when we have reached an optimal solution for the covering array problem. Instead such a 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. For each feasible solution $S \in \Sigma$, we define 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. The algorithm then generates a set of sequences (or

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

Table 3. $VCA(27; 2, 3^9 2^2, \{CA(27; 3, 3^3)^3\})$

Markov chains) of trials. If $c(S') \leq c(S)$ then the transition is accepted. If the transition results in a feasible solution S' 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 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 1.

The idea of allowing a move to a worse solution helps avoid being stuck in a bad configuration (*a local optimum*), while continuing to make progress. Sometimes we know the cost of an optimal solution and can stop the algorithm when this is reached. Otherwise we stop the algorithm when insufficient progress is being made (as determined by an appropriately defined *stopping condition*). In this case the algorithm is said to be *frozen*.

Simulated annealing has been used by Nurmela and Östergård [9], for example, to construct covering designs which have a structure very similar to covering arrays. It has also been used by Stardom and Cohen, *et al.* to generate minimal fixed strength covering arrays for software testing [6, 10].

In the simulated annealing algorithm the current feasible solution is an approximation S to a covering array in which certain t-subsets are not covered. The cost function is based on the number of t-subsets that are not currently covered. A covering array itself will have cost 0. In the variable strength array, the cost is 0 when (i) all of the t-subsets are covered and (ii) for each covering array of strength t' in C, all t'-subsets are covered. A potential transition is made by selecting one of the K-sets belonging to S and then replacing a random point in this K-set by a random point not in the K-set. We calculate only the change in numbers of t-sets that will occur with this move. Since the subsets of C are disjoint, any change we make can affect the overall coverage of the VCA and at most one of the subsets of C. This means that the extra work required for finding variable strength arrays over fixed strength arrays does not grow in proportion to the number of subsets of higher strength.

We keep the number of blocks (test cases) constant throughout a simulated annealing run and use the method described by Stardom to determine a final array size [10]. We start with a large random array and then bisect our array repeatedly until we find a *best* solution.

4.2. Program Parameters

Good data structures are required to enable the relative cost of the new feasible solution to be calculated efficiently, and the transition (if accepted) to be made quickly. We build an exponentiation table prior to the start of the program. This allows us to approximate the transition probability value using a table lookup. We use ranking algorithms from [8] to hold the values of our t-sets. This allows us to generalize our algorithms for different strengths without changing the base data structure. When calculating the change in cost for each transition, we do not need to recalculate all of the t-sets in a test case, but instead only calculate the change (t-1 subsets).

A constant is set to determine when our program is frozen. This is the number of consecutive trials allowed where no change in the cost of the solution has occurred. For most of our trials this constant has been set to 1,000.

The cooling schedule is very important in simulated annealing. If we cool too quickly, we freeze too early because the probability of allowing a worse solution drops too quickly. If we cool too slowly or start at too high a temperature, we allow too many poor moves and fail to make progress

Therefore, if we start at a low temperature and cool slowly we can maintain a small probability of a bad move for a long time allowing us to avoid a frozen state, at the same time continuing to make progress. We have experimented using fixed strength arrays compared with known algebraic constructions (see [6]). We have found that a starting temperature of approximately 0.20 and a slow cooling

VCA	{**C**}	**Min N**	**Max N**	**Avg N**
$VCA(2, 3^{15}, \{C\})$	$\emptyset$	16	17	16.1
	$CA(3, 3^3)$	27	27	27
	$CA(3, 3^3)^2$	27	27	27
	$CA(3, 3^3)^3$	27	27	27
	$CA(3, 3^4)$	27	27	27
	$CA(3, 3^5)$	33	33	33
	$CA(3, 3^4)$, $CA(3, 3^5)$, $CA(3, 3^6)$	33* 34	35	34.8
	$CA(3, 3^6)$	33* 34	35	34.9
	$CA(3, 3^7)$	41	42	41.4
	$CA(3, 3^9)$	50	51	50.8
	$CA(3, 3^{15})$	67	69	67.6
$VCA(2, 4^3 5^3 6^2, \{C\})$	$\emptyset$	36	36	36
	$CA(3, 4^3)$	64	64	64
	$MCA(3, 4^3 5^2)$	100	104	101
	$CA(3, 5^3)$	125	125	125
	$CA(3, 4^3)$, $CA(3, 5^3)$	125	125	125
	$MCA(3, 4^3 5^3 6^1)$	171	173	172.5
	$MCA(3, 5^1 6^2)$	180	180	180
	$MCA(3, 4^3 5^3 6^2)$	214	216	215
$VCA(2, 3^{20} 10^2, \{C\})$	$\emptyset$	100	100	100
	$CA(3, 3^{20})$	100	100	100
	$MCA(3, 3^{20} 10^2)$	304	318	308.5

Table 4. Table of sizes for variable strength arrays after 10 runs (We have omitted the parameter N in our notation for covering arrays in this table due to space limitations.)

* The minimum values for these VCA's were found during a separate set of experiments

factor, α, of between 0.9998 and 0.99999 every 2,500 iterations works well. Using these parameters, the annealing algorithm completes in a "reasonable" computational time on a PIII 1.3GHz processor running Linux. For instance, the first few VCA's in Table 4, complete in seconds, while the larger problems, such as the last VCA in Table 4, complete within a few hours. The delta in our cost function is counted as the change in t-sets from our current solution. Since we can at any point make changes to both the base array and one of the higher strength arrays, these changes are added together.

As there is randomness inherent in this algorithm, we run the algorithm multiple times for any given problem.

5. Results

Table 4 gives the minimum, maximum and average sizes obtained after 10 runs of the simulated annealing algorithm for each of the associated VCA's. Each of the 10 runs uses a different random seed. A starting temperature of .20 and a decrement parameter of .9998 is used in all cases. In two cases a smaller sized array was found during the course of our overall investigation, but was not found during one of these runs. The numbers are included in the table as well and are labeled with an asterisk, since these provide a previously unknown bound for their particular arrays. In each case we show the number of tests required for the base array of strength two. We then provide some examples with variations on the contents of C. Finally we show the arrays with all of the columns involved in strength three coverage. We have only shown examples using strength two and three, but our methods should generalize for any strength t.

What is interesting in Table 4 is that the higher strength sub-arrays often drive the size of the final test suite. Such is the case in the first and second VCA groups in this table. We can use this information to make decisions about how many components can be tested at higher strengths. Since we must balance the strength of testing with the final size of the test suite we can use this information in the design process.

Of course there are cases where the higher strength subsets do not determine the final test suite size since the number of test cases required is a combination of the number of levels *and* the strength. In the last VCA group in Table 4 the two components each with 10 levels require a minimum of 100 test cases to cover all pairs. In this case we can cover all of the triples from the 20 preceding columns with the same number of tests. In such cases, the quality of the tests can be improved without increasing the number of test cases. We can set the strength of the 20 preceding columns to the highest level that is possible without increasing the test count.

Both situations are similar in the fact that they allow us to predict a minimum size test suite based on the fixed level sub-arrays. Since there are better known bounds for fixed strength arrays we can use this information to drive our decision making processes in creating test suites that are both manageable in size while providing the highest possible interaction strengths.

6. Conclusions

We have presented a combinatorial object, the variable strength covering array, which can be used to define software component interaction tests and have discussed one computational method to produce them. We have presented some initial results with sizes for a group of these objects. These arrays allow one to guarantee a minimum strength of overall coverage while varying the strength among disjoint subsets of components. Although we present these objects for their usefulness in testing component based software systems they may be of use in other disciplines that currently employ fixed strength covering arrays.

The constraining factor in the final size of the test suite may be the higher strength sub-array. We can often get a second level of coverage for almost no extra cost. We see the potential to use these when there is a need for higher strength, but we cannot afford to create an entire array of higher strength due to cost limitations.

Where the constraining factor is the large number of levels in a set of fields at lower strength, it may be possible to increase the strength of sub-arrays without additional cost, improving the overall quality of the tests.

Another method of constructing fixed strength covering arrays is to combine smaller arrays or related objects and to fill the uncovered t-sets to complete the desired array [4]. We are currently experimenting with some of these techniques to build variable strength arrays. Since the size of a VCA may be dependent on the higher strength arrays, we believe that building these in isolation followed by annealing or other processes to fill in the missing lower strength t-sets will provide fast and efficient methods to create optimal variable strength arrays.

Acknowledgments

Research is supported by the Consortium for Embedded and Internetworking Technologies and by ARO grant DAAD 19-1-01-0406. Thanks to the Consortium for Embedded and Internetworking Technologies for making a visit to ASU possible.

References

[1] J. Bach. James Bach on risk-based testing In *STQE Magazine*, Nov/Dec,1999.

[2] L. Brownsword, T. Oberndorf and C. Sledge. Developing new processes for COTS-based systems. *IEEE Software*, 17(4):48–55, 2000.

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

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

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

[6] 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 Sofware Engineering (ICSE 2003)*, 2003, pp. 38-48 , Portland.

[7] 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.

[8] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms, Generation, Enumeration and Search*. CRC Press, Boca Raton, 1999.

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

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

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

[12] A. W. Williams. Determination of test configurations for pair-wise interaction coverage In *Proc. Thirteenth Int. Conf. Testing Communication Systems*, 2000, pp. 57–74.