
Biased covering arrays for progressive ranking and composition of Web Services

Renée C. Bryce*

Department of Computer Science,
University of Nevada at Las Vegas, Las Vegas, NV 89154, USA
E-mail: reneebruce@cs.unlv.edu
*Corresponding author

Yinong Chen and Charles J. Colbourn

Department of Computer Science and Engineering,
Arizona State University,
Tempe, AZ 85257, USA
E-mail: yinong@asu.edu E-mail: Charles.Colbourn@asu.edu

Abstract: Service-oriented computing is a new software development paradigm that allows application builders to choose from many available services. The challenges are to efficiently determine which services are the most appropriate to combine into an application based on concerns such as functionalities, licensing costs, and known reliability. Group testing has addressed some issues on how to select a small subset of candidate services. We expand upon the group testing methodology with a greedy algorithm that generates biased covering arrays for interactive testing of the services selected by group testing.

Keywords: biased covering arrays; covering arrays; group testing; mixed-level covering arrays; pair-wise interaction testing; reliability; service-oriented architecture; web services; WS.

Reference to this paper should be made as follows: Bryce, R.C., Chen, Y. and Colbourn, C.J. (2007) 'Biased covering arrays for progressive ranking and composition of Web Services', *Int. J. Simulation and Process Modelling*, Vol. 3, Nos. 1/2, pp.80–87.

Biographical notes: Renée C. Bryce is an Assistant Professor in at the University of Nevada at Las Vegas. She completed her PhD at Arizona State University. Her research interests are software engineering and design, including software testing and human-computer interaction.

Yinong Chen received his PhD from the University of Karlsruhe, Germany. He is currently a Senior Research Scientist at Arizona State University. His research areas include dependable computing and software engineering. He is interested in applying fault-tolerant computing techniques to Web Services (WS), distributed systems, and embedded systems.

Charles J. Colbourn earned his PhD in Computer Science in 1980 from the University of Toronto. He was Dorothean Professor of Computer Science at the University of Vermont prior to joining Arizona State University as a Professor in 2001. He is the author of 260 journal papers, four books and numerous conference papers. He serves on six editorial boards for international journals in combinatorics. In 2004, he was awarded the Euler Medal for Lifetime Achievement in Research by the Institute for Combinatorics and its Applications. His current research concerns combinatorial aspects of mobile ad hoc networks, optical networks, communications and coding, and software testing.

1 Introduction

Web Services (WS) represent an emerging technology that has the promise to become a standard in computing. Almost all major computer corporations, such as IBM, Microsoft, Sun Microsystems, Oracle and SAP, are moving in this direction and have started to offer WS and relevant development tools. Major Government agencies, such as the National Science Foundation, NASA and Department of Defense in the USA, are creating programs

to fund web computing-based research. The movement is gaining momentum, creating numerous challenges and opportunities. WS differ from traditional software in many respects. When planning a new web service, original development is not the only option. Existing WS can be searched, located, and remotely invoked to provide the required functionality, or a part thereof. A new web service can be composed dynamically at runtime, completely or partially, using existing WS available over the internet.

To ensure the trustworthiness of the new composite service, the constituent services found over the internet must be tested in the composite service. Thus, the traditional ‘test before delivery’ pattern is no longer sufficient for the new paradigm. WS must be tested not only before, but also after, being published and composed into another web service at runtime. Another difference is that service providers may provide the user interface and functionality only. They may not be willing to provide the source code. Thus, testing can only be based on the specification and the interface definition of the WS.

1.1 Web service testing

To ensure interoperability, various standards have been defined to regulate the specification, for example, Business Process Execution Language for Web Services (BPEL4WS), Web Ontology Language for Services (OWL-S), WS Description Language (WSDL), Unified Modeling Language (UML), publication and search for example Universal Description, Discovery and Integration (UDDI), communication (HTTP, XML) and invocation (SOAP – Simple Object Access Protocol) (Ankolekar et al., 2002; Curbera et al., 2002, 2003; Milanovic and Malek, 2004; Tsai et al., 2002, 2003). When planning a new WS, the business decision concerns whether to purchase existing WS to compose the new service, or to pay developers to write a new one from scratch for this purpose, which can also be offered on the internet for profit. The former approach has the advantage of rapid development; the latter approach may be more cost effective in the long term. As a result, a large number alternative WS may exist for any given specification.

A number of studies of WS testing have been made. In Bloomberg (2002), WS testing technology is divided into three phases. In phase one, WS were essentially tested like ordinary software. In phase two (2003–2005), the following features were included in testing: publishing, finding, and binding capabilities of WS; the asynchronous capabilities of WS; the SOAP intermediary capability; and the quality of services. In phase three (2004 and beyond), the following features were tested: dynamic runtime capabilities, WS orchestration testing, and WS versioning. Without actually constructing a new service, web service orchestration defines a process that coordinates execution order of existing WS using flow control commands such as sequence, parallel, and switches (Milanovic, 2006). On the other hand, WS composition is a process of constructing a new service that binds (writing the activation addresses of the selected WS into the code of the composed WS). Orchestration testing can be used as a proof-of-concept testing before the service is constructed. In Davidson (2002) distinguished between two kinds of WS testing: internet-based and intranet-based testing. He also listed several testing techniques including: proof-of-concept testing, functional testing, regression testing, load/stress testing, and monitoring. In De (2003), it was suggested that WS testing should include: basic WS functionality;

SOAP messages; WSDL files; publishing, finding, binding capabilities of the Service-Oriented Architecture (SOA); asynchronous capabilities; the SOAP intermediary capability; the quality of service; dynamic runtime capabilities; SOAP and WS interoperability; and performance and load testing.

This work extends previous work with a two step process. Section 1.2 describes how Group Testing is applied to narrow down the number of prospective candidates for each web service that may later be combined into a final composite service. Section 1.3 then introduces how interaction testing may be applied to generate economically sized test suites that further evaluate the candidate WS.

1.2 Selecting and rating Web Services (WS) with group testing

Previous studies address testing individual WS, but offer no effective solution for testing a large number of WS with the same specification. A group testing technique, originally developed for testing a large number of blood samples (Du and Hwang, 2000) and later for software regression testing, is an attractive solution to address this problem. In blood group testing, large samples of blood are pooled and mixed together. A new sample is taken from the mixed blood for contamination test. If the new sample is safe, the entire group is safe. If the new sample is contaminated, smaller sample groups will have to be formed for group testing.

In Tsai et al. (2004a), a Web Services Group Testing (WSGT) technique was first proposed to test a large number of WS in a manner more efficient than individual testing. The main idea of WSGT can be outlined as follows. Assume the new service to be composed consists of n component WS: WS_1, WS_2, \dots, WS_n . For component WS_i , there are v_i alternative WS available. Thus, there are $k_1 \times k_2 \times \dots \times k_n$ different possibilities in total to compose the new service. Since k_i is in general a large number (as discussed above) it is considerably difficult to test all combinations. Group testing applies the same inputs to the WS that meet the same specification, executes the WS simultaneously, and votes the results. The WS whose outputs agree with the majority output are considered to have produced a correct output. Each WS is rated according to the rate of the correct outputs. Voting complex output data is not a trivial task. Tsai et al. (2005a) proposed an efficient scheme that can vote non-numerical and numerical data with deviation and with complex data structure.

1.3 Interaction testing of selected Web Services (WS)

Once group testing has narrowed down the number of options to consider using for each web service and rated them, they need to be tested. This paper proposes an efficient algorithm to test composite services, and to help to ultimately find a practical solution. The main idea is to make use of the group testing results that rank each

set of the component WS (Tsai et al., 2004b), and then progressively select the best component WS to construct the composite service by using interaction testing.

The construction of test suites is a challenging problem when composite services consist of many individual WS and the combinatorial growth of interactions among the WS to test becomes too costly. WS group testing must determine a set of composite services to construct that reveals not only errors in the individual WS, but also those that result from interactions among the WS. One expects that interactions among a small number of the chosen WS may compromise correctness or performance. Hence, once candidate WS are identified for further test, the test suite constructed provides coverage of the potential interactions. In particular, for a threshold t , for every t of the types of services in the composite service, and for every selection from the candidates for these services, some test must be a composite service in which these t services arise from this selection. When the number of services and the number of candidates for each service is small, exhaustive testing suffices; but when either is large, judicious selections are needed to minimise the size of the test suite. To handle the combinatorial explosion, covering arrays are used. These have been proposed for interaction testing for software (Cohen et al., 2003a, 2003b; Colbourn et al., 2004), and effective algorithms exist for their construction (Chateaufneuf et al., 1999; Cheng et al., 2003; Cohen et al., n.d., 2003c). For WSGT, a variant of covering arrays is needed to treat the different confidence levels of trust for the individual services; in essence, different coverage is needed depending upon these confidence levels. Greedy methods, such as Colbourn et al. (2004), can be adapted to construct test suites with such coverage.

The rest of the paper is organised as follows. Section 2 describes a combinatorial abstraction of interaction test suites to covering arrays, and outlines their features and limitations in WS testing. Section 3 then develops a greedy strategy for test suite construction that employs rankings of the constituent WS. Section 4 gives a small example and experimental data; Section 5 concludes with a discussion of the potential use of this method.

2 Biased covering arrays

The underlying representation of the tests described in this paper is a biased covering array, which is derived from the definition of a covering array. This section begins with formal definitions and background, then describes algorithmic considerations in literature that can be applied to constructing biased covering arrays, and finally describes an actual method for rapidly building biased covering arrays.

2.1 Background

Definition: A covering array, $CA_\lambda(N; t, k, v)$, is an $N \times k$ array. In every $N \times t$ subarray, each t -tuple occurs at least λ times within the N rows of the covering array.

In our application, t is the *strength* of the coverage of interactions, k is the number of components (degree), and v is the number of symbols for each component (order). In all of our discussions, we treat only the case when $\lambda = 1$, i.e., that every t -tuple must be covered at least once.

To expand upon this definition, consider the following example of three WS as shown in Table 1. For each of the three WS, there are three available implementations. As noted earlier, these implementations may have different licensing costs and confidence levels.

Table 1 Three Web Services (WS) each have three different implementations to select from

<i>Rain forecast WS</i>	<i>Temperature forecast WS</i>	<i>Wind forecast WS</i>
0	3	6
1	4	7
2	5	8

A covering array for pair-wise interaction testing of this data in Table 1 is shown in Table 2 and requires only nine rows.

Table 2 A covering array for pair-wise interaction testing requires only nine tests for the input shown in Table 1

<i>Test no.</i>	<i>Rain forecast WS</i>	<i>Temperature forecast WS</i>	<i>Wind forecast WS</i>
1	0	3	6
2	1	3	7
3	0	4	8
4	2	5	6
5	1	5	8
6	2	4	7
7	0	5	7
8	1	4	6
9	2	3	8

The covering array is fundamental when all factors have an equal number of levels. However, the number of levels (or options) for every factor (web service type) may not always be the same. Then the mixed-level covering array can be used.

Definition: A mixed level covering array, $MCA_\lambda(N; t, k, (v_1, v_2, \dots, v_k))$, is an $N \times k$ array. Let $\{i_1, \dots, i_t\} \subseteq \{1, \dots, k\}$, and consider the subarray of size $N \times t$ obtained by selecting columns i_1, \dots, i_t of the MCA . There are

$$\prod_{i=1}^t v_i$$

distinct t -tuples that could appear as rows, and an MCA requires that each appear at least once.

Definition: The Covering Array Number (CAN) is the size of the smallest possible covering array that can be constructed for an input. Finding the smallest possible solution is NP-hard (Colbourn et al., 2004).

Some interactions are more important to cover than are others; the covering array does not distinguish this, as it assumes that all rows will be run as tests. When only some rows are to be used as tests, certain tests are more desirable than others.

Suppose that there are k different services that are to be composed; call the services S_1, \dots, S_k . For each service, the repository may contain a number of choices. So suppose that, for each i , service S_i has ℓ_i choices $S_{i,1}, \dots, S_{i,\ell_i}$.

Among these choices we may include not only the trusted ones, but also all choices that have been deemed to be of some value. For each choice $S_{i,j}$, we assume that a numerical value between 0 and 1 is available as an output from the group testing described previously. We assume that every service τ_i for service S_i has a trust value t_{i,τ_i} ; this value is between 0 (the lowest ranking of trust) and 1 (the highest).

Each *test* within a test suite consists of an assignment to each service (factor) S_i of a value τ_i with $1 < \tau_i < \ell_i$. Evidently some tests involve more trusted selections than do others, so our 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 which can take on values of 0–1, where 1 is the strongest weight. Specifically, the trust attached to choosing τ_i for S_i and τ_j for S_j together is $t_{i,\tau_i} t_{j,\tau_j}$.

We compute the *benefit* of an n th test (in isolation) to be the amount of new weight covered in the test:

$$\beta_n = \sum_{i=1}^k \sum_{j=i+1}^k t_{i,\tau_i} t_{j,\tau_j}.$$

Every pair covered by the test contributes to the total benefit, according to the reported value of trust associated with each pair that has not been included in a previous test. Two trusted services contribute a greater amount to the benefit when either is more trusted. Indeed if we were forced to run just one test, the one with largest benefit would select the most trusted option for each service, as expected. However in general we are prepared to run many tests. Consider a test suite consisting of many tests. Now, 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 can 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.

Definition: 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. For instance, if the tests are executed sequentially, the

largest benefit is incurred in the earlier tests. The subsequent amount of benefit either stays the same or decreases in the last rows.

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 (Colbourn et al., 2004). Worse yet, we rarely know the value of ℓ 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.

We first describe algorithms for creating traditional covering arrays and then introduce an algorithm that adapts for application to a biased covering array.

Biased covering arrays differ from covering arrays in that selections for factor values have individual benefits (trust), and the objective is to cover not all pairs of choices, but rather to cover the pairs of trusted choices.

Several types of algorithms for generating covering arrays exist. These include mathematical constructions (Cheng et al., 2003; Hartman, 2002; Williams and Probert, 1996), heuristic search (Cohen et al., 2003a, 2003b, 2003c, n.d.) and greedy methods (Bryce et al., 2005; Colbourn et al., 2004). Of these different types of construction methods, greedy algorithms from (Bryce and Colbourn, 2007; Bryce et al., 2005) are the most natural fit with the problem of a biased covering array because tests are incrementally built one-row-at-a-time. A biased covering array covers all pairs (or t -tuples); however, not all tests in a biased covering array are run in practice. Testers may only run a subset of a test suite and want to execute the tests with the largest benefit first. Satisfying this concern is a strength of greedy algorithms. A greedy algorithm selects values in attempt to provide as much coverage as possible in each iteration. Earlier rows have the largest amount of coverage, but the amount of coverage packed into subsequent rows often slowly diminishes as there is less left to be covered. This is exemplified in Section 3.

2.2 Covering arrays to account for weight

Biased covering arrays are a variation of the more studied objects, covering arrays. We begin this section with a review of how to construct covering arrays with a greedy approach and then expand on how a greedy method can adapt to generate biased covering arrays.

The generation of covering arrays and mixed-level covering arrays fall into a greedy framework (Bryce et al., 2005) and can be described at a high level. The overall goal in constructing a covering array is to create a two-dimensional array in which all t -tuples associated with a specified input are covered. Using a greedy approach, this collection is built one row at a time by fixing each factor with a level value. The order in which factors are fixed may vary, as can the scheme for choosing levels (values that can be selected for a factor). A row may be selected from multiple candidates. When more than one candidate is permitted, several rows are constructed and one is chosen to

add to the covering array. Once all t -tuples have been covered, the covering array is complete. The goal is to construct the smallest covering array possible, so if random selections occur, the covering array can be regenerated numerous times to select the best result.

An instantiation for a biased covering array may include any number of repetitions, candidates, or factor ordering rules. However, at a minimum, new level selection rules are needed to adapt to the criteria of weighted coverage. The instantiation introduced next, modifies the density formulas presented in the Deterministic Density Algorithm (Colbourn et al., 2004) to work with weights.

2.3 An algorithm for biased covering arrays

We consider the construction of a test suite with k factors, adapting an algorithm that falls into the greedy framework, the Deterministic Density Algorithm (Colbourn et al., 2004). The number of levels for factor i is denoted by ℓ_i . The benefit of covering a pair of selections, τ_i for S_i and τ_j for S_j , is $t_{i,\tau_i} t_{j,\tau_j}$; the incremental benefit is the same when the pair is covered for the first time in this test, and zero if the coverage occurs in an earlier test. For factors i and j , we define the *total benefit* β_{ij} to be

$$\sum_{a=1}^{\ell_i} \sum_{b=1}^{\ell_j} t_{i,a} t_{j,b},$$

while the *remaining benefit* ρ_{ij} is the same sum, but of the incremental benefits. Define the *local density* to be $\delta_{i,j} = (\rho_{i,j} / \beta_{ij})$. In essence, δ_{ij} indicates the fraction of benefit that remains available to accumulate in tests to be selected. We define the *global density* to be

$$\delta = \sum_{1 \leq i < j \leq k} \delta_{i,j}.$$

At each stage, we endeavour to find a test whose incremental benefit is at least δ .

To select such a test, we repeatedly fix a value for each factor, and update the local and global density values. At each stage, some factors are *fixed* to a specific value, while others remain *free* to take on any of the possible values. When all factors are fixed, we have succeeded in choosing the next test. Otherwise, select a free factor S_s . We have

$$\delta = \sum_{1 \leq i < j \leq k} \delta_{i,j},$$

which we separate into two terms:

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

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

Write $\rho_{i,s,\sigma}$ for the ratio of the sum of incremental benefits of those pairs involving some level of factor f_i , and level σ of factor S_s to the sum of (usual) benefits of the same set of pairs. Then rewrite the second summation as

$$\sum_{\substack{1 \leq i \leq k \\ i \neq s}} \delta_{i,s} = \frac{1}{\ell_s} \sum_{\sigma=1}^{\ell_s} \sum_{\substack{1 \leq i \leq k \\ i \neq s}} \rho_{i,s,\sigma}.$$

We choose σ to maximise

$$\sum_{\substack{1 \leq i \leq k \\ i \neq s}} \rho_{i,s,\sigma}.$$

It follows that

$$\sum_{\substack{1 \leq i \leq k \\ i \neq s}} \rho_{i,s,\sigma} \geq \sum_{\substack{1 \leq i \leq k \\ i \neq s}} \delta_{i,s}.$$

We then fix factor S_s to have value σ and update the local densities setting $\delta_{i,s}$ to be $\rho_{i,s,\sigma}$. In the process, the density has not been decreased (despite some possible – indeed necessary – decreases in some local densities).

We iterate this process until every factor is fixed. The factors could be fixed in *any order at all*, and the final test has density at least δ . Of course it is possible to be greedy in the order in which factors are fixed. Rows are sequentially generated until all pairs, or t -tuples, are covered.

This method ensures that each test selected furnishes at least the *average* incremental benefit. This may seem to be a modest goal, and that one should instead select the test with maximum incremental benefit. However, even when all trust values are equal, it is NP-hard to select such a test (see Colbourn et al., 2004).

3 Comparison of weighting

Both the original and weighted algorithms based on density were implemented for comparison. The modification to the original density method to incorporate weights includes earlier coverage of the pairs that are deemed most important. To compare the two approaches using density, consider the input shown in Table 3. Although a larger problem would illustrate different features of the method, we have chosen to employ data from real WS. This data is for a simulation to predict weather conditions to determine whether to launch a satellite at a specific location at a specified date and time (Tsai et al., 2004c). Three component services were used for

- rain forecast
- temperature forecast
- wind forecast.

For each of these three components, ten options of acceptable reliability were available. To rank the ten available options for each of the three WS, group testing was used to identify the reliability of each (Tsai et al., 2004c). This data is shown in Table 3 as services 0 through 9. In order to model the inclusion of less reliable services (omitted from Tsai et al. (2004c)), we adjoined five more options for each service, having reliabilities of 0.2, 0.15,

0.1, 0.001 and 0.001. These five additional options for each of the services are labelled as services 10 through 14.

Table 3 Weights of three Web Services (WS) having 15 options each with the first ten options and reliability ratings from group testing output and five additional options and reliability ratings inserted for example

Service	Rain forecast WS rating	Temperature forecast WS rating	Wind forecast WS rating
0	0.86	0.99	0.86
1	0.78	0.99	0.95
2	0.84	0.96	0.91
3	0.57	0.98	0.87
4	0.91	0.97	0.91
5	0.78	0.97	0.96
6	0.78	0.99	0.89
7	0.76	0.98	0.88
8	0.68	0.98	0.91
9	0.68	0.99	0.86
10	0.2	0.2	0.2
11	0.15	0.15	0.15
12	0.1	0.1	0.1
13	0.001	0.001	0.001
14	0.001	0.001	0.001

Before developing the weighted density algorithm, we introduced the idea of simply retaining the best options for each, in order to focus on the most reliable services. In Figure 1, four such selections are shown, restricting to consider only the top 3, 5, 8 and 10 top rated options, and then building a covering array. The graph shows that the coverage of weight plateaus first when restricted to only the top 3 factors; followed by those with the top 5, 8 and 10 factors shown in the graph respectively. Evidently when very few tests are to be run, this example indicates that simply restricting to the most reliable options for each can be sensible. Indeed the weight accumulated within the first few tests is essentially the same whether or not such a restriction is in effect. As the graphs show, however, when more tests are to be run it is not desirable to restrict the number of options. Moreover, since this example has only three factors, it does not demonstrate one important phenomenon. It can happen, when the number of options for a factor is restricted to only few, that pairs are repeatedly covered when new pairs could have been covered. For instance, if there is a large difference in the weight among levels for a factor, those with the largest weights will repeatedly be incorporated into the earliest tests while those with the smallest weights are least important and are filled in at the end of the test suite.

In Figure 1, a small variation in the amount of weight covered can be noted. This results from the unweighted nature of the algorithm used. In order to assess the benefits of taking weights into account in determining density, Figure 2 shows the extra weight covered after each test by the weighted density algorithm in comparison to the

unweighted one, each using the first ten options for each web service included in Table 3.

Figure 1 Cumulative weight covered retaining only the best 3, 5, 8 and 10 options

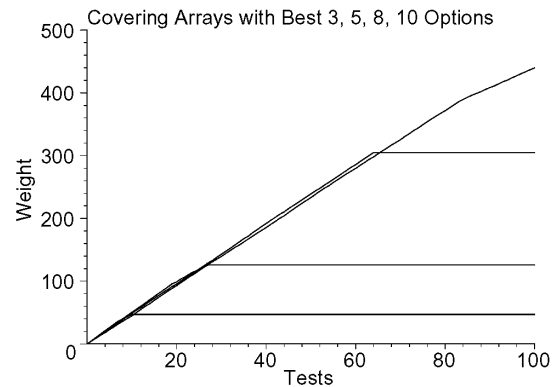
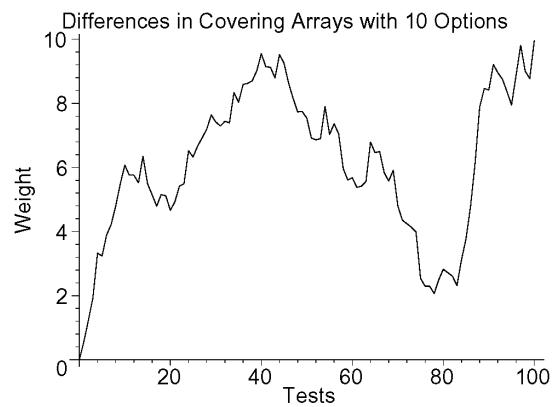


Figure 2 Cumulative extra weight covered using weighted density vs. unweighted density



In order to prioritise the execution of the tests, the weighted density approach should be used. This becomes more pronounced when less reliable services are included. To illustrate this, we treat the three factors with 15 values each. Figure 3 shows that the weight accumulated by the unweighted algorithm covers weight slower than the weighted version of generation for this example.

In this case, the extra weight covered in the first few tests by weighted density is striking, as shown in Figure 4.

Figure 3 Cumulative weight covered using weighted density and unweighted density

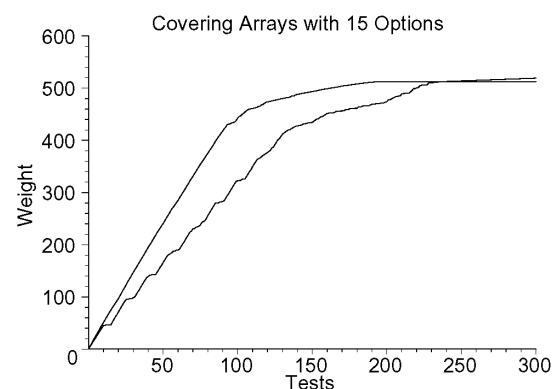
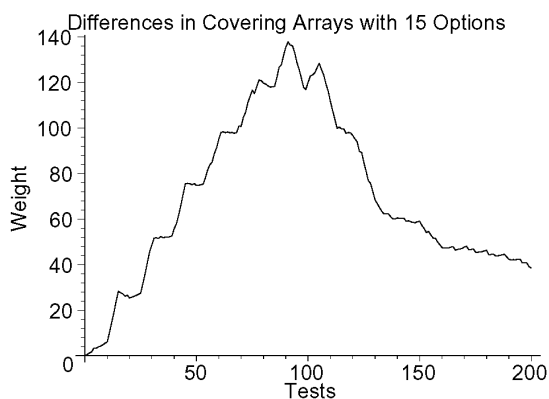


Figure 4 Cumulative extra weight covered using weighted density vs. unweighted density

4 Conclusions

Test generation for testing WS in a service-oriented architecture is a challenging part of providing a trusted repository. A main problem is that, while many versions may be of interest to users, the maintenance of many different versions becomes problematic when many different alternative WS for a given functional specification are provided. Restricting attention to the most trusted services and restricting to few different types of services, one could perform ‘exhaustive’ testing. Permitting more services, each with a few trusted versions, one could use covering arrays to provide test suites. However, retaining all potentially useful versions while concentrating on the most trusted, we must select tests that stress the most trusted versions but do not ignore the rest. Here we have generalised the notion of covering arrays so that the number of versions for each service need not be known in advance, and so that each version can have an associated level of trust. We have shown a greedy method based on densities that determines a sequence of tests to be performed; each test attempts to make the best incremental improvement in a ‘benefit’ measurement that rewards a test for coverage of pairs of versions of services that are trusted. In an example, we show that a new algorithm can provide test suites in a prioritised order. This is only a small component of the design of a repository, addressing testing needs for those with many versions of many services. Nevertheless, it provides an easily implemented method that appears to treat such testing problems effectively.

Acknowledgements

Research supported by the Consortium for Embedded and Inter-Networking Technologies (CEINT).

References

- Ankolekar, A., Burstein, M., Hobbs, J.R., Lassila, O., Martin, D., McDermott, D., McIlraith, S.A., Narayanan, S., Paolucci, M., Payne, T. and Sycara, K. (2002) ‘DAML-S: web service description for the semantic web’, *Proc. International Semantic Web Conference (ISWC)*, October, pp.348–363.
- Bloomberg, J. (2002) ‘Web services testing: beyond SOAP’, *Zap-Think LLC*, September 2002, <http://www.zapthink.com>.
- Bryce, R.C. and Colbourn, C.J. (2007) ‘The density algorithm for pairwise interaction coverage’, *Journal of Software Testing, Verification, and Reliability*, to appear.
- Bryce, R.C., Colbourn, C.J. and Cohen, M.B. (2005) ‘A framework of greedy methods for constructing interaction test suites’, *Proc. 27th International Conference on Software Engineering (ICSE2005)*, May, pp.146–155.
- Chateaneuf, M.A., Colbourn, C.J. and Kreher, D.L. (1999) ‘Covering arrays of strength three’, *Designs, Codes and Cryptography*, Vol. 16, May, pp.235–242.
- Cheng, C., Dumitrescu, A. and Schroeder, P. (2003) ‘Generating small combinatorial test suites to cover input-output relationships’, *Proceedings of the Third International Conference on Quality Software (QSIC '03)*, Dallas, November, pp.76–82.
- Cohen, M.B., Colbourn, C.J. and Ling, A.C.H. (n.d.) ‘Constructing strength three covering arrays with augmented annealing’, *Discrete Mathematics*, to appear.
- Cohen, M.B., Colbourn, C.J., Collofello, J.S., Gibbons, P.B. and Mugridge, W.B. (2003a) ‘Variable strength interaction testing of components’, *Proc. 27th Annual International Computer Software and Applications Conference (COMPSAC 2003)*, Dallas, TX, November, pp.413–418.
- Cohen, M.B., Colbourn, C.J., Gibbons, P.B. and Mugridge, W.B. (2003b) ‘Constructing test suites for interaction testing’, *Proc. International Conf. Software Engineering (ICSE03)*, Portland, OR, May, pp.38–48.
- Cohen, M.B., Colbourn, C.J. and Ling, A.C.H. (2003c) ‘Augmenting simulated annealing to build interaction test suites’, *Proc. IEEE Int. Symp. Software Reliability Eng. (ISSRE 2003)*, Denver, CO, November, pp.394–405.
- Colbourn, C.J., Cohen, M.B. and Turban, R.C. (2004) ‘A deterministic density algorithm for pairwise interaction coverage’, *Proceedings of the International Conference on Software Engineering (SE 2004)*, Innsbruck, Austria, February, pp.245–252.
- Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N. and Weerawarana, S. (2002) ‘Unraveling the web services web: an introduction to SOAP, WSDL and UDDI’, *IEEE Internet Computing*, Vol. 6, No. 2, March, pp.86–93.
- Curbera, F., Khalaf, R., Mukhi, N., Tai, S. and Weerawarana, S. (2003) ‘The next step in web services’, *Commun. ACM*, Vol. 46, No. 10, October, pp.29–34.
- Davidson, N. (2002) *Testing Web Services*, <http://www.webservices.org>, October.
- De, B. (2003) *Web Services – Challenges and Solutions*, WIPRO white paper, <http://www.wipro.com>.

- Du, D.Z. and Hwang, F. (2000) *Combinatorial Group Testing and its Applications*, 2nd ed., World Scientific, Singapore.
- Hartman, A. (2002) 'Software and hardware testing using combinatorial covering suites', *Haiifa Workshop on Interdisciplinary Applications of Graph Theory, Combinatorics, and Algorithms*, June.
- Milanovic, N. and Malek, M. (2004) 'Current solutions for web service composition', *IEEE Internet Computing*, November, pp.51–59.
- Milanovic, N. (2006) 'Service engineering design patterns', *Second IEEE International Symposium on Service-Oriented System Engineering (SOSE'06)*, pp.19–26.
- Tsai, W.T., Chen, Y., Cao, Z., Bai, X., Huang, H. and Paul, R. (2004b) 'Testing web services using progressive group testing', *Advanced Workshop on Content Computing*, Zhenjiang, China, November, pp.314–322.
- Tsai, W.T., Chen, Y., Paul, R., Liao, N. and Huang, H. (2004a) 'Cooperative and group testing in verification of dynamic composite web services', *Workshop on Quality Assurance and Testing of Web-Based Applications, in Conjunction with COMPSAC*, September, pp.170–173.
- Tsai, W.T., Chen, Y., Zhang, D. and Huang, H. (2005a) 'Voting multi-dimensional data with deviations for web services under group testing', *Proceedings of the 4th International Workshop on Assurance in Distributed Systems and Networks (ADSN), in Conjunction with ICDCS-25*, pp.65–71.
- Tsai, W.T., Paul, R., Cao, Z., Yu, L., Saimi, A. and Xiao, B. (2003) 'Verification of web services using an enhanced UDDI server', *Proc. IEEE WORDS*, January, pp.131–138.
- Tsai, W.T., Paul, R., Wang, Y., Fan, C. and Wang, D. (2002) 'Extending WSDL to facilitate web services testing', *Proc. IEEE HASE*, October, pp.171, 172.
- Tsai, W.T., Zhang, D., Chen, Y., Huang, H., Paul, R. and Liao, N. (2004c) 'A software reliability model for web services', *Proceedings of 8th IASTED International Conference on Software Engineering and Applications*, September, pp.144–149.
- Williams, A.W. and Probert, R.L. (1996) 'A practical strategy for testing pair-wise coverage of network interfaces', *Seventh Intl. Symp. on Software Reliability Engineering*, October, White Plains, New York, pp.246–254.

Bibliography

- Bryce, R.C. and Colbourn, C.J. (2005) 'Test prioritization for pairwise coverage', *Proc. ICSE 2005: Workshop on Advances in Model-Based Software Testing (A-MOST)*, Minneapolis, Minnesota, May.
- Burner, M. (2004) *Service Orientation and its Role in Your Connected Systems Strategy*, Microsoft White Paper, July.
- Coehn, F. (2003) *Testing Web Services*, McGraw-Hill Osborne Media.
- Colbourn, C.J. (2004) 'Combinatorial aspects of covering arrays', *Le Matematiche (Catania)*, Vol. 58, pp.121–167.
- Myerson, J. (2002) *Testing for SOAP Interoperability*, February, <http://www.webservicesarchitect.com>.
- Tsai, W.T., Paul, R., Yu, L., Wei, X. and Zhu, F. (2005b) 'Rapid pattern-oriented scenario-based testing for embedded systems', in Yang, H. (Ed.): *Software Evolution with UML and XML*, Idea Group Publishing, Hershey, Pennsylvania, pp.222–262.
- WS-I (2005) *WS-I Usage Scenarios, Supply Chain Management Use Case Model*, Sample Application Supply Chain Management Architecture, <http://www.ws-i.org/>.
- Yilmaz, C., Cohen, M.B. and Porter, A. (2004) 'Covering arrays for efficient fault characterization in complex configuration spaces', *Proceedings Intl. Symp. on Software Testing and Analysis (ISSTA2004)*, July, pp.45–54.