

Progressive Ranking and Composition of Web Services Using Covering Arrays

Charles J. Colbourn, Yinong Chen, and Wei-Tek Tsai
Department of Computer Science and Engineering,
Arizona State University,
Tempe, AZ, 85287-8809,
U.S.A.
{colbourn, yinong.chen, wtsai}@asu.edu

Abstract

Major computer companies and government agencies are adopting Web Services (WS) technology. Web services must ensure interoperability and security, and be reliable and trustworthy. Consumers must make runtime decisions based on an intelligent and unbiased evaluation. We propose one component of a test suite generator to address this critical issue. The goal is to transform the labor-intensive software development process into a largely automated web services development process. The method operates within a service-oriented architecture, with the models and tools implemented as services that can be added, removed and replaced at runtime. We propose a generic greedy algorithm based on covering arrays for application in this environment.

“Web services are not yet widely used because of security concerns. But there’s an even bigger roadblock waiting just down the road – it’s called trust. The big issue is “Will the service work correctly every time when I need it?” As yet few are thinking about the issues of testing and certification. We suggest that testing and certification of Web services is not business as usual and that new solutions are needed to provide assurance that services can really be trusted.” (CBDi Forum, <http://searchwebservices.techtarget.com>, 11 July 2002).

1. Introduction

Web Services (WS) represent an emerging technology that has the promise to become the standard of computing. Almost all major computer corporations (including IBM, Microsoft, Sun Microsystems, Oracle, and SAP) are moving in this direction; they have started to offer web services and relevant development tools. Major government agencies, such as the National Science Foundation (NSF) and

Department of Defense (DoD) in the United States, are creating programs to fund web computing-based research. The movement is gaining momentum, creating numerous challenges and opportunities. Web services differ from traditional software in many respects. When planning a new web service, original development is not the only option. Existing web services can be searched, located, and remotely invoked to provide the required functionality, or a part thereof. A new service can be composed dynamically and at runtime, completely or partially, using existing available over the Internet. Thus, the traditional “test before delivery” pattern is no longer sufficient. Web services must be tested not only before, but also after, being composed into another web service at runtime. Another difference is that service providers may provide only the user interface and functionality. They may not be willing to provide source code. Thus, testing can only be based on the functional specification and the interface definition of the web services.

To ensure interoperability, standards have been defined to regulate the specification (for example, BPEL4WS, DAML-S, OWL-S, WSDL, WSFL, UML), publication and search (UDDI), invocation (HTTP, XML), and communication (SOAP) [1, 15, 16, 21, 26, 27]. When planning a new web service, the business decision concerns whether to purchase existing web services to compose the new service, or to pay developers to write a new one from scratch and sell it 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 web services may exist for any given specification. The primary question that this paper addresses is how to test the large number of available web services to construct trustworthy new services.

A number of studies of testing have been made. In [2], web services testing technology is divided into three phases. In phase one, web services were essentially tested like ordinary software. In phase two (2003-2005), the following

features were included in testing: publishing, finding, and binding capabilities of web services; the asynchronous capabilities of web services; the SOAP intermediary capability; and the quality of services. In phase three (2004 and beyond), the following features were tested: dynamic runtime capabilities, web services orchestration testing, and web services versioning. Without actually constructing a new service, web services orchestration defines a process that coordinates execution order of existing web services using flow control commands such as sequence, parallel, and switch [22]. On the other hand, web services 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 [17], Davidson distinguished between two kinds of web services 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 [18], it was suggested that web services testing should include: basic web services functionality; SOAP messages; WSDL files; publishing, finding, binding capabilities of an SOA; asynchronous capabilities; the SOAP intermediary capability; the quality of service; dynamic runtime capabilities; SOAP and web services interoperability; and performance and load testing. In [7], Clune and Chen suggested that both clients and service providers should be involved in web services testing, and many issues must be addressed during web services development including: security, interoperability, UDDI registration, and performance considerations.

These studies address testing individual web services, but offer no effective solution for testing large number of web services with the same functionality. A group testing technique, originally developed for testing a large number of blood samples [19] and later for software regression testing, is an attractive solution to address this problem. In [25], a WS Group Testing (WSGT) technique was first proposed to test a large number of web services 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 web services: WS_1, WS_2, \dots, WS_n . For component WS_i , there are k_i alternative web services 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 obviously impossible to test all combinations. This paper proposes an innovative and efficient algorithm to test the composite service, and find a practical solution. The main idea is to make use of the group testing results that rank each set of the component web services, respectively [25], and then progres-

sively select the best component web services to construct the composite service.

The construction of test scripts is a challenging combinatorial problem when composite services consist of many individual web services, as testing all combinations becomes prohibitive. Web services group testing must determine a set of composite services to construct that reveals not only errors in the individual web services, but also those that result from interactions among the web services. One expects that interactions among a small number of the chosen web services may compromise correctness or performance. Hence, once candidate web services 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 test generation suffices; but when either is large, judicious selections are needed to minimize the size of the test suite. To handle the combinatorial explosion, covering arrays are used. These have been proposed for interaction testing for software [14, 11, 12], and effective algorithms exist for their construction [5, 6, 9, 10]. For WSGT, a variant of covering arrays to treat the different confidence levels for the individual services is needed; in essence, different coverage is needed depending upon these confidence levels. Greedy methods, such as [14], can be adapted to construct test suites with such coverage.

The rest of the paper is organized as follows. Section 2 describes a combinatorial abstraction of interaction test suites to covering arrays, and outlines their features and limitations in web services testing. Section 3 then develops a greedy strategy for test suite construction that employs rankings of the constituent web services. Section 4 gives a small example; this is not intended to provide practical validation of the method, rather to assist in understanding it. Section 5 concludes with a discussion of the potential use of this method.

2. Covering Arrays

A *covering array*, $CA_\lambda(N; t, k, v)$, is an $N \times k$ array. In every $N \times t$ subarray, each t -tuple occurs at least λ times. In our application, t is the *strength* of the coverage of interactions, k is the number of components (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.

This combinatorial object is fundamental when all factors have an equal number of levels. However, software systems are typically not composed of compo-

nents (*factors*) that each have exactly the same number of parameters (*levels*). Then the mixed-level covering array can be used. A *mixed level covering array*, $MCA_\lambda(N; t, k, (v_1, v_2, \dots, v_k))$, 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.

Covering arrays have been explored from a mathematical viewpoint, and have substantial applications in testing; see [13, 20] for extensive surveys. For web services testing, however, covering arrays address some but not all of the needs. Their best feature is perhaps that with an economy of tests they ensure that all interactions of a small number of component selections are evaluated. Nevertheless, their drawback is that as defined they require knowledge of all choices in advance, and treat all choices equally. In check-in of a web service, the user has a certain level of trust in each of the components already present in the repository. Indeed the repository itself maintains a ranking of the component selections. Check-in of a new web service requires that the service be tested along with the most trusted services already present, since a user can be expected to prefer services that are trusted (indeed this is one definition of trust). One way to proceed is to restrict our attention to the most trusted choices for each candidate service, and to employ tests from a covering array to evaluate each of these combinations. This fails to guide the user who chooses less trusted components for reasons of their own, and tends to extend further trust to selections that are already trusted. A service that obtains a low ranking initially in such a scheme may be effectively eliminated from further testing despite offering a service that some users want. Employing covering arrays with only the most trusted selections for each service also results in redundant testing, since numerous combinations may be repeated in the tests of the test suite.

Our goal is therefore to cover all t -way interactions among the most trusted component selections, but at the same time include less trusted selections once the interactions among the most trusted have been covered. We develop this more formally next, focussing on the case when $t = 2$ (i.e. when pairs are covered).

2.1 A Coverage Model

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 $c_{i,1}, \dots, c_{i,\ell_i}$. 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 $c_{i,j}$, we assume that a numerical value between 0 and 1 is available as a measure of the

trustedness of this selection for service S_i ; a value of 1 indicates complete trust, while a value of 0 indicates no trust.

A *test* consists of an assignment to each service (factor) S_i of a value τ_i with $1 \leq \tau_i \leq \ell_i$. 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, we compute the *benefit* of a test (in isolation) as

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

Every pair covered by the test contributes to the total benefit, according to the trustedness of the selections for the two services in the pair. Two trusted services contribute a greater amount to the benefit than when either is less 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 can 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. These 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.

3. Building a Test Suite

We consider the construction of a test suite with k factors, adapting an algorithm from [14]. 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 $\delta_{i,j} = \frac{\rho_{i,j}}{\beta_{i,j}}$. In essence, $\delta_{i,j}$ 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 covering 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 maximize $\sum_{\substack{1 \leq i \leq k \\ i \neq s}} \rho_{i,s,\sigma}$. 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 , set $v_s = 1$, 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.

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 [14]).

3.1 Discussion

The algorithm described expresses the preference for more trusted services, but once these are tested it proceeds progressively to test services in decreasing order of trustfulness. More importantly, it tests *pairs* of trusted services first when both are highly trusted, next when one is trusted and the other less so, and finally for two less trusted services. This affords a method for testing that allows one to perform the tests sequentially, and terminate once sufficient confidence is achieved.

For check-in of web services, let us imagine that a provider releases a new version of a service. Our goal is then to ascertain compatibility (and hence trustfulness) of

this new version with other services in the repository, focussing on the more trusted ones (since these are preferentially selected by users). In order to assess compatibility, one can use the algorithm proposed to generate a test suite for the selections of other services, to be bound dynamically with the service being registered and then tested. This enables us to determine the operation of the new version in conjunction not just with the (most trusted) services in the repository, but also with pairs of such services. Since testing can proceed ordered by current trust levels, the process can be run until sufficient confidence is obtained either to accept or to reject the new version for inclusion in the repository. When accepted, we must also calculate a level of trustfulness for the new version. We could simply assess its compatibility based on its success in integration with other services through the basic acceptance tests. However this does not address the question of how trusted it is relative to the already registered versions of the same service. To assess this, we can use the algorithm proposed once again, this time incorporating the new version of the service among the options for that service, and generating tests for all services. In this manner, we obtain a “head-to-head” comparison of the new service with existing ones.

Once accepted, the repository can again be tested as a whole with the new version included; the new tests then adjust the ranking of services according to new trust levels. In this way, a version that was heretofore less trusted can increase in trustfulness and become competitive for user selection once again.

This brief discussion is not intended to detail how test results can be used to determine trustfulness; rather our focus is on a general method to construct tests that provide coverage of trusted services in assessing new versions for check-in and validation.

4. An Illustrative Example

In order to illustrate the behavior of the method, in this section we outline an example. This is not intended to be a validation of the method in practice; rather it is intended to demonstrate the types of test suites that the method produces. Our example has twenty factors (components) in total; six have seven levels each, four have five, four have four, and six have three. The components are listed in this order in the test suite, and the levels for each are indicated by numbers from $\{0, \dots, \ell - 1\}$, where ℓ is the number of choices for the component. In selecting a test suite, it can happen that no matter which level is selected for a factor, no new pair is covered. When that occurs, the entry is recorded as a dash (–), indicating a “don’t care” position.

Tests here are generated and presented in order of decreasing importance. Each test is selected so as to cover pairs involving the most trusted components that have not

previously been covered. To illustrate this, we require trust values to assign to each selection for each component. Determining these trust values is beyond the scope of the work here. Therefore, for the purposes of illustration only, we adopt a simple formula for assigning fictitious trust values. We simply set the trust for the j th selection of a component with ℓ choices to be $\frac{(\ell-j)^2(\ell+j)}{\ell^3}$. These trust values range from 0 to 1 for each component; again, they are not meant to reflect actual trust values, but rather to furnish a differentiation among selections so that the method can exploit this preference. For convenience, trust values are chosen so that selection i is more trusted than selection j (for the same component) exactly when $i < j$.

Test #	Test
0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1	1 1 1 1 1 1 0 1 1 1 0 0 1 1 0 0 0 1 0 1
2	2 2 2 0 2 1 1 0 0 1 1 1 0 0 1 1 1 0 1 0
3	0 1 0 2 1 2 1 2 2 0 1 1 1 0 1 0 1 1 0 0
4	1 0 2 1 0 2 2 1 0 2 1 1 0 1 1 1 0 0 1 1
5	3 3 1 0 3 0 1 0 1 2 2 2 1 1 0 1 1 0 1 0
6	0 0 3 3 2 3 2 1 1 0 0 2 2 0 0 0 1 1 1 1
7	2 2 0 1 3 0 0 1 2 0 0 0 2 2 1 1 0 2 0 1
8	2 0 1 2 0 1 2 2 2 1 2 2 0 2 0 2 2 0 2 2
9	1 1 3 0 1 0 2 0 0 3 2 1 2 2 2 2 0 2 0 2
10	3 3 2 3 1 3 0 2 3 2 1 0 0 0 2 0 0 1 0 2
11	1 3 0 2 2 4 0 3 1 0 0 1 0 1 2 2 0 2 1 2
12	0 2 4 4 0 3 1 3 1 3 0 0 1 1 0 0 1 1 2 0
13	0 4 1 1 4 2 3 0 3 1 0 0 2 0 0 1 2 2 1 0
14	4 4 3 2 4 0 0 2 0 1 1 2 1 1 1 1 0 1 0 1
15	3 1 0 3 0 1 3 3 0 2 2 1 2 3 1 1 2 0 0 1
16	4 1 2 4 3 4 3 1 0 0 2 0 0 0 0 0 0 0 2 0
17	2 3 4 0 2 2 0 1 2 3 2 3 1 2 2 0 1 1 0 1
18	1 2 1 3 3 3 1 0 0 1 3 1 3 0 1 2 0 1 0 1
19	3 0 4 1 1 4 1 1 3 3 1 2 1 2 0 1 1 2 1 2
20	0 3 5 5 3 1 2 3 3 0 1 0 3 3 1 2 2 0 0 0
21	0 5 2 2 5 0 3 0 2 2 0 0 1 2 2 2 1 1 1 2
22	2 1 3 4 2 0 3 2 3 2 3 0 3 1 1 0 2 2 1 2
23	4 4 4 3 0 5 1 0 1 0 1 3 0 2 1 0 0 0 2 0
24	5 4 0 4 4 3 2 1 2 1 3 1 1 3 0 1 1 0 2 1

Table 1. Test suite, beginning

Consider the tests enumerated in Table 1; these are the first 25 tests from a test suite of 78 tests in which every pair of component selections is covered. Each row provides a test number, followed by a vector of 20 integers, one for each component. The i th entry in the test indicates the selection made in this test for the i th component. The first test chooses selection 0 for each component. This is consistent with the specification that the most trusted selection for each of the components is the first one (the one indexed by ‘0’). The second test could have been chosen so that the pairs covered are completely disjoint from those covered in the first test (as the method in [14] would). This

would involve making selection 0 for at most one component in the test. However here the second test covers some pairs already covered in the first test. The reason is simple. By choosing to cover pairs that involve greater trust, the algorithm determines that the inclusion of selections that are most trusted (i.e. those selections indicated by ‘0’ in our example), it is reasonable to cover fewer uncovered pairs in order to cover more “important” ones.

Among the first 25 tests generated, consider the number of times each selection is made for the first component. Selection 0 is made seven times, selection 1 five times, selection 2 five times, selection 3 four times, selection 4 three times, selection 5 once, and selection 6 is not chosen at all. For the last component, having only three selections, we find that selection 0 is made ten times, selection 1 is made nine times, and selection 2 is made six times. This illustrates two important features. As expected, more trusted selections are made with higher frequency. More importantly, when a component has more selections, the method introduces less trusted selections as more and more tests are generated. This is the central idea that distinguishes this from a method that restricts *a priori* to a small number of trusted selections for each component. Indeed the method can be applied to generate further tests until *all* pairs of component selections are covered.

This example serves to show the behavior of the algorithm, but does not provide any real validation of its use in testing. Part of our current research is to examine how to choose trust values so that the tests prove most effective at identifying defects.

5. Conclusions

Test generation for testing web services 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 services 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 generalized 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 proposed 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 cov-

erage of pairs of versions of services that are trusted. 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 treats such testing problems effectively.

Acknowledgments

Thanks to Renée Bryce for helpful discussions. Research supported by the Consortium for Embedded and Inter-Networking Technologies.

References

- [1] A. Ankolekar et al., DAML-S: Web Service Description for the Semantic Web, *Proc. International Semantic Web Conference (ISWC)*, 2002, pp. 348-363.
- [2] J. Bloomberg, Web services testing: Beyond SOAP, ZapThink LLC, Sep 2002, <http://www.zapthink.com>
- [3] R. C. Bryce, C. J. Colbourn, and M. B. Cohen. A framework of greedy methods for constructing interaction test suites. In *Proc. 27th International Conference on Software Engineering (ICSE2005)*, page to appear, May 2005.
- [4] M. Burner, Service Orientation and Its Role in Your Connected Systems Strategy, Microsoft White Paper, July 2004.
- [5] M.A. Chateaneuf, C.J. Colbourn, and D.L. Kreher, Covering arrays of strength three, *Designs, Codes and Cryptography* 16 (1999), 235-242.
- [6] C. Cheng, A. Dumitrescu and P. Schroeder, Generating small combinatorial test suites to cover input-output relationships, *Proceedings of the Third International Conference on Quality Software (QSIC '03)*, Dallas, November 2003, pp. 76-82.
- [7] J. Clune and L. Chen, Testing Web Services: Methods for ensuring server and client reliability <http://www.sys-con.com/websphere/>.
- [8] F. Coehn, Testing Web Services, McGraw-Hill Osborne Media, 2003.
- [9] M.B. Cohen, C.J. Colbourn, and A.C.H. Ling, Constructing Strength Three Covering Arrays with Augmented Annealing, *Discrete Mathematics*, to appear.
- [10] M.B. Cohen, C.J. Colbourn, and A.C.H. Ling, Augmenting simulated annealing to build interaction test suites, *Proc. IEEE Int Symp Software Reliability Eng (ISSRE 2003)*, Denver CO, 2003, pp. 394-405.
- [11] M.B. Cohen, C.J. Colbourn, J.S. Collofello, P.B. Gibbons, and W.B. Mugridge, Variable Strength Interaction Testing of Components, *Proc. 27th Annual International Computer Software and Applications Conference (COMPSAC 2003)*, Dallas TX, November 2003, pp. 413-418.
- [12] M.B. Cohen, C.J. Colbourn, P.B. Gibbons, and W.B. Mugridge, Constructing test suites for interaction testing, *Proc. International Conf. Software Engineering (ICSE03)*, Portland OR, May 2003, pp. 38-48.
- [13] C.J. Colbourn. Combinatorial Aspects of Covering Arrays. *Le Matematiche (Catania)*, to appear.
- [14] C.J. Colbourn, M.B. Cohen, and R.C. Turban, A Deterministic Density Algorithm for Pairwise Interaction Coverage, *Proceedings of the International Conference on Software Engineering (SE 2004)*, Innsbruck, Austria, February 2004, pp. 245-252.
- [15] F. Curbera et al., Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI, *IEEE Internet Computing*, 6,2 (2002), 86-93.
- [16] F. Curbera et al., The Next Step in Web Services, *Commun. ACM*, 46,10, (2003), 29-34.
- [17] N. Davidson, Testing Web Services, <http://www.webservices.org>, October 2002.
- [18] B. De, Web Services - Challenges and Solutions, WIPRO white paper, 2003, <http://www.wipro.com>.
- [19] D. Z. Du and F. Hwang, *Combinatorial Group Testing And Its Applications*, World Scientific, 2nd edition, 2000.
- [20] A. Hartman, Software and Hardware Testing Using Combinatorial Covering Suites, in: *Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications*, Kluwer Academic Publishers, to appear.
- [21] N. Milanovic and M. Malek, Current Solutions for Web Service Composition, *IEEE Internet Computing*, Nov/Dec, 2004, 51-59.
- [22] N. Milanovic and M. Malek, Verifying Correctness of Web Services Composition, *Proceedings of the 11th Infotest*, Budva, Montenegro 2004
- [23] J. Myerson, Testing for SOAP Interoperability, <http://www.webservicesarchitect.com>, Feb 2002.

- [24] W. T. Tsai, Y. Chen, Z. Cao, X. Bai, H. Huang, R. Paul, Testing Web Services Using Progressive Group Testing, *Advanced Workshop on Content Computing*, Zhenjiang, China, November 2004.
- [25] W.T. Tsai, Y. Chen, R. Paul, N. Liao, and H. Huang, 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 2004.
- [26] W.T. Tsai, R. Paul, Z. Cao, L. Yu, A. Saimi, and B. Xiao, Verification of Web Services Using an Enhanced UDDI Server, *Proc. IEEE WORDS*, 2003, pp. 131-138.
- [27] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang, Extending WSDL to Facilitate Web Services Testing, *Proc. IEEE HASE*, 2002, pp. 171-172.
- [28] W. T. Tsai, R. Paul, L. Yu, X. Wei, and F. Zhu, Rapid Pattern-Oriented Scenario-Based Testing for Embedded Systems, in *Software Evolution with UML and XML*, edited by H. Yang, to appear.
- [29] W. T. Tsai, L. Yu, R. Paul, T. Liu, and A. Saimi, Developing Adaptive Test Frameworks for Testing State-Based Embedded Systems, *Proc. IDPT*, 2002.
- [30] WS-I: <http://www.ws-i.org/>, WS-I Usage Scenarios, Supply Chain Management Use Case Model, Sample Application Supply Chain Management Architecture.
- [31] C. Yilmaz, M.B. Cohen, and A. Porter. Covering arrays for efficient fault characterization in complex configuration spaces. *Proceedings Intl. Symp. on Software Testing and Analysis (ISSTA2004)*, 2004, pp. 45-54.