W. T. Tsai, X. Wei, Y. Chen, "A Robust Testing Framework for Verifying Web Services by Completeness and Consistency Analysis," IEEE International Workshop on Service-Oriented System Engineering (SOSE), Beijing October 2005, pp.151-158.

A Robust Testing Framework for Verifying Web Services by Completeness and Consistency Analysis

W. T. Tsai, X. Wei, Y. Chen, R. Paul*
Computer Science and Engineering Department
Arizona State University, Tempe, AZ 85287-8809, U.S.A
*Department of Defense, Washington DC, U.S.A

Abstract

This paper presents a specification-based robust testing framework for Web services. Web services testing is done by 1) extracting condition and event combinations from the Web service specification, 2) ensuring that these combinations are consistent with each other; 3) performing completeness analysis to identify all the missing condition and event combinations; 4) Identifying the locations where updates are needed to maintain the completeness and consistency.; 5) emphasizing on robustness testing by generating positive as well as negative test cases. An efficient algorithm called Covering Scenario Generation is proposed to identify the locations where incompleteness and inconsistency exist. The algorithm is based on the min-terms of Boolean expressions that combine multiple conditions into a single checkable item. The proposed algorithm has been experimented with several large industrial applications and the results indicated that the proposed algorithm is robust and scalable to large applications. A case study is designed to illustrate the design and testing process.

Keywords: Web services, Web services testing, test case generation, completeness and consistency checking.

1. Introduction

Web services (WS) have emerged in e-commerce and e-business applications in the recent years. Current WS are based on UDDI server that provides directory services similar to the telephone yellow book. A UDDI server is not responsible for the quality of the services it refers. Thus, the trustworthiness of WS present a major concern for the users and a barrier to widening the application of WS. Traditional dependability techniques, such as correctness proof, model checking, fault-tolerant computing, testing, and evaluation, can be used to improve the trustworthiness of WS. However, these techniques have to be redesigned to handle the dynamic and the open platform features of WS. This paper exploits WS-based test case generation, testing, and verification techniques that can lead to trustworthy WS.

A number of studies and attempts have been made to address the WS testing and verification problems.

In [5] Davidson distinguished between two kinds of WS testing: Internet-based and Intranet-based WS testing. He also listed several testing techniques for WS including: proof-of-concept testing, functional testing, regression testing, load/stress testing, and monitoring.

WS testing should include: basic WS functionality; SOAP messages; WSDL files; publishing, finding, binding capabilities of an SOA; asynchronous capabilities of WS; the SOAP intermediary capability; the quality of service of WS; dynamic runtime capabilities; SOAP and WS interoperability; and WS performance and load testing.

In [4], Clune and Chen suggested that both clients and service providers must be involved in WS testing, and many issues must be addressed during WS development including: security, interoperability, UDDI registration, and performance considerations. Similarly, in [8], Myerson stated that all three parties of WS including clients, providers, and brokers, need to be involved in WS testing. These discussions support the collaborative testing concept we propose for WS testing.

Traditional white-box testing techniques cannot be directly applied in WS testing in that WS providers usually do NOT publish source code. Instead, they only publish WS specifications, or sometimes together with WS executables. More and more commercialized WS specifications are represented in OWL-S (Ontology Web Language for Service). As a part of the DARPA Agent Makeup Language program, OWL-S facilitates WS specification with a core set of markup language constructs for describing the properties and capabilities of WS in unambiguous and computer-interpretable form. OWL-S supports the automation of WS tasks including automated WS discovery, invocation, interoperation, composition, and execution monitoring. It includes three main parts: the service profile for advertising and discovering services; the process model, which gives a detailed description of a service's operation; and the

grounding, which provides details on how to interoperate with a service, via messages.

Based on these studies, we proposed the topology-based rapid systematic positive and negative testing approach that assures the trustworthiness of WS by completeness and consistency (C&C) analysis, pattern identification technique, rigorous positive and negative test case generation. These techniques not only perform positive testing that verify the required functionality but also perform negative testing that investigates the robustness of WS under irregular inputs. Negative testing is particular important for WS, because a WS can be composed of WS from different service providers without the availability of the source codes.

C&C studies often address both completeness and consistency simultaneously. However several recent studies address primarily one aspect only and address the other aspect only if it is detected while addressing the first issue. For example, Heitmeyer, Labaw and Kiskis primarily address the consistency issues within SCR (Software Cost Reduction) [18]. However, detecting inconsistencies in an SCR specification may also lead to incompleteness discovery. Thus, although the technique in [18] is called a consistency checker, it has attributed to both C&C checking.

C&C analysis techniques presented in this paper are developed on an event-driven modeling and specification language [10], where system states are defined as predicates or conditions of actors. After WS specification are represented in this language, the C&C algorithm is applied to analyze all the combinations of system conditions as well as internal and external events to identify if the system specified is complete with respect to all these combinations. In other words, if the system is considered complete by the C&C algorithm, the system specified can respond to all the specified events, internal, and external events, in any system states as defined by system conditions. Different from completeness analysis, consistency analysis checks if the requirements explicitly specify the condition-event-action in correct way. Condition combinations represent system states. For each specific condition combination, the system should react properly upon the associated event occurrence. Consistency analysis is to check whether the condition-event-action leads to an intended state. The C&C analysis discussed in this paper primarily addresses the completeness issues.

The behavior specification of real-time systems often can be classified into patterns that include pre-conditions (or causes) and post-conditions (or effects) with timing constraints as an option [10]. Three kinds of patterns are especially important in WS testing: scenario pattern (SP), verification pattern (VP) and robustness pattern (RBP). An SP is defined as a specific temporal pattern or a cause-effect relation that can be used in representing a number of requirements. A timeline is used to depict the temporal relations of the events involved in the SP. A VP is a pre-defined verification mechanism that can be used to verify a group of positive scenarios with identical control flow. Similarly, a RBP is a pre-defined verification mechanism that can be used to verify a group of negative scenarios with identical control flow. It shares the same logic as corresponding VP, except that the key-event is an abnormal event. It verifies if the abnormal event handlers work correctly. A VP or RBP is described as follows:

- Description and purpose
- Class diagram (for the requirement representation)
- Verification State Machine (representing the verification logic)

With the experience, we classified and identified eight kinds of scenario patterns, which are found to cover 95% of requirements of an implantable Medical Device (IMD), and 100% of requirements of example Car Alarm System (CAS) [10]. By mapping scenario patterns into verification patterns, and reusing test cases generated from verification patterns, the cost of writing test cases can be greatly reduced. This significantly reduces testing cost because it is estimated that 70% of effort during the integration testing is spent on debugging test cases rather than developing the test cases and performing actual tests. Pattern-oriented development and testing mechanism can therefore greatly reduce system development budget because testing cost usually takes up 70% cost of the whole budget.

Swiss Cheese model [11][12][13] is a new approach to select test data based on the Boolean expressions extracted from the conditions and decisions in the program under test. The test data selection model is based on the Hamming distance (HD) and the Boundary Degree (BD) defined on the cells of the Karnaugh maps corresponding to the Boolean expressions. Each cell corresponds to a test data and the pair (HD, BD) represents the fault detection potency of the test data. This model extends the current boundary-based test data selection models to a generic form that includes the distance to the boundary and the degree of the boundaries. A generic algorithm is developed to find the hierarchy of the test data ranked by their fault detection potency of topological structure. A tool has been developed for automated test data generation. The model and tool has been applied in a number of industrial projects. The

experiment data confirm the effectiveness of the new test selection model.

2. The Development Process

Software testing can base on source code, executable (binary code), and specification. White box testing requires all three resources while black box testing can base on the latter two resources. Obviously, the white box testing makes use of more information and can better ensure the trustworthiness. With the source code available, the shadowy code (unnecessary and unwanted code) can be easily found. With specification-based black box testing, one can easily generate test case set to verify the required functionality (positive testing). However, it is difficult to detect malicious or Trojan faults that can be activated if a particular input pattern, normally an invalid input, is entered (negative testing). For WS testing, negative testing is paramount, because WS is an open platform that allows WS developed by different providers to be integrated into a composite service.

Our development and testing process considers three cases: (1) Only specification is available; (2) Specification and executable are available; and (3) Specification, executable, and source code are available. In case (1), it is assume that the WS specification in OWL-S is available, the process includes the following four major steps, as also shown in Figure 1:

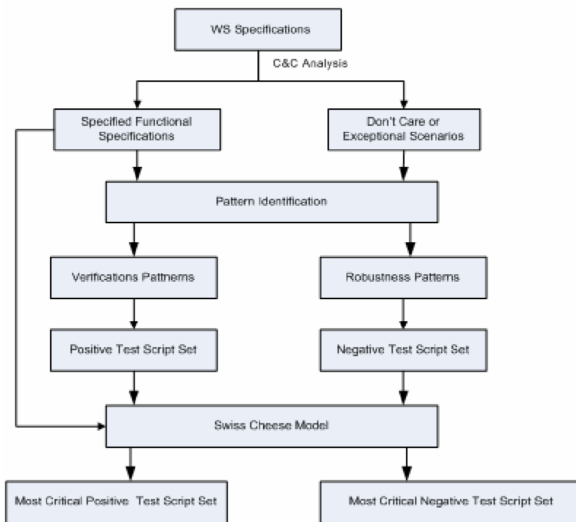

Figure 1. WS Testing Process

1. Perform C&C analysis on the WS specification and identify those incomplete and inconsistent conditions and constraints. After iteratively supplementing missing conditions, two kinds of conditions may remain unspecified: 1) don't care conditions due to physical constraints; 2) exception handling.
2. Categorize specified scenarios into verifications patterns and unspecified scenarios into robustness patterns.
3. Automatically generate positive and negative test cases using pattern-oriented test case generation approach.
4. Use Swiss Cheese (SC) model to select most critical positive and negative test cases.

3. WS Verification Using Completeness and Consistency Analysis

3.1 Completeness and Consistency Introduction

Because the proposed completeness algorithm analyzes all the Conditions and Events combinations (called CE combinations), the number of CE combinations in a given WS specifications can be rather large, and thus often a large number of incompleteness can be identified (See Section 3.4). Fortunately, even though the number of incompleteness identified can be large, the number of scenarios that need to be revised is often small, in fact sometimes only a few scenarios need be to be revised as the proposed algorithm can minimizes the number of scenarios that need to be revised (see Section 3.3 and 3.4 for details). The principal idea of the algorithm is similar to identifying the minimum representation of a Boolean expression using Karnaugh Map [14]. Thus, the proposed techniques have been effective in solving large applications including:

- A Best Buy Stock (BBS) Web service
- Several large real-time distributed command-and-control NCW (Network-Centric Warfare) systems for the US Department of Defense (DoD);
- A large real-time mission-critical process control for semiconductor manufacturing for Intel corporation; and
- A large real-time high-availability communication processor for Motorola.

The proposed algorithm has further been implemented into a tool that can run efficiently on these WS. And in most cases, the tool takes only a few minutes (running on a PC) to identify the incompleteness as well as the scenarios that need be revised.

3.2 Completeness and Consistency Process

We have developed a specific scenario/ACDATE model and it has been used to specify a commercialized Best Buy Stock (BBS) Web service [11][13] and several large DoD (US Department of Defense) real-time

command-and-control systems [10]. One of the C&C issue in this model is that it is necessary to examine all the CE combinations. We developed a heuristic and iterative algorithm to identify the missing CE combinations by first partitioning the systems into groups of components so that the components in different groups are independent of each other. Then testing only needs to ensure that all the CE combinations in each group are completely covered. In this way, the number of CE combinations can be greatly reduced. Figure 2 shows the overall C& C analysis process.

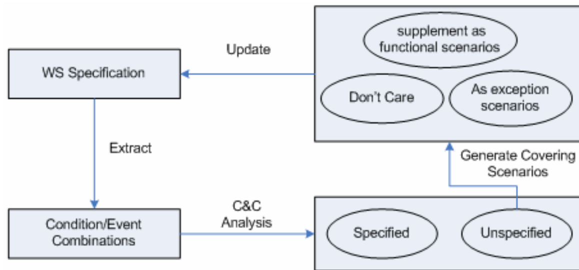

Figure 2. Process of C&C Analysis

1. Parse each scenario and extract the combinations of Conditions and Events (CE): The CE combinations are partitioned into groups of components where each component does not interact with or related to the components in other groups. Two scenarios are independent of each other if there is no way for them to interact or influence each other. For example, two scenarios that share a common condition is considered related, and the "related" relationship is transitive. By exhaustively examining the transitive relationships, one can determine if two scenarios are independent of each other.
2. Perform C&C analysis on CE combinations: Once the CE combinations are extracted, the consistency analysis is performed on the combinations first, and then the completeness analysis is performed to identify those missing CE combinations.
3. Identify the covering scenarios to eliminate those missing CE combinations. An algorithm is developed to aggregate a large number of missing CE combinations into a smaller set of equivalent CE combinations, which we call **covering scenarios**, to facilitate the elaboration on the missing scenarios. The algorithm is described in Section 3.3.
4. Classify each covering scenario into three categories: 1) incorporate it as a functional scenario; 2) treat it as an exception with an exception handler; or 3) treat it as an a don't care item due to physical constraints. In the first case, the covering scenario is indeed an intended behavior but missed in the specification. In the second case the covering scenario is not intended and should be masked out in the specification. In the third case, it can be either included or excluded.
5. Revise scenario specification using the C&C analysis results: Use the results in step 5 to revise the scenario specification.

3.3 Covering Scenario Generation Algorithm

The completeness analysis will first identify the missing CE combinations. This step is straightforward because any CE combinations that are not specified in the scenario specification are considered missing. The key is to combine those missing CE combinations so that the user can address the missing CE combinations rapidly because the number of missing CE combinations can be huge. It is really troublesome for a user to read each CE combination and address them one-by-one. This section introduces an algorithm that generates *reduced* number of scenario set that covers all the missing combinations. The scenarios are called covering scenarios, because each of them may represent more than one basic CE combination, (each condition in the basic CE combination is either "1" or "0"). For example, binary string "1101××001" is a covering scenario, which covers 4 basic CE combinations by extending the two "don't care" or '××' conditions.

Two CE combinations can be combined into a covering scenario if and only if they satisfy the following conditions:

1. They handle the same event;
2. They have the same number of conditions, and the same digits in each position except one position *p*.
3. At the position p, one of the combinations takes "0", and the other takes "1".

For example, the condition combination 0×10 can be combined with 0×11 and get covering scenario 0×1×, but cannot be combined with 0×01. The reason that 0×10 cannot be combined with 0×01 is because they have two different digits in 3rd and 4th positions. In order to form the combination: 0×××, those two condition combinations need to be combined with two more condition combinations: 0×00 and 0×11.

The next step is to identify covering scenarios based on the missing CE combinations. The proposed algorithm uses the following strategy to identify the candidate scenarios:

1. Search the combinations so that they have the same number of "don't care" digits;
2. Then search the combinations so that they have difference in a single digit only.

According to the above strategy, the missing CE combinations are divided into $k*k$ groups, where k is the number of conditions. Members in a group $G(r, t)$ have the same number r of "1"s, and same number t of "×",

where $0 \le r, t \le k$. Each member CE_i in a group has an attribute representing by a tuple T_{CEi} = (x, y), such that *x* is obtained by converting "×"s in the combination string into "0"s and calculating the binary expression into a decimal integer; and *y* is obtained by converting "1"s into "0"s, "×" into "1"s, and calculating the binary expression into a decimal integer. Within each group, sort the members according to the values of *y*, and if *y* values are the same, sort according to the values of *x*.

Algorithm:

```
For (n = 0; n <= K; n++) //Combine covering scenario groups in the increasing order of the
                           //number of "Don't care" Items
{
  For (i = 0; i <= K-n; i++) //For the fixed n, combine covering scenario groups (I, n) in
                              //the increasing order of the number of '1's
  {
    For  (j = 0; j < Sizeof_Group(i,n); j++)  //Fixed I, n, combine covering
                                               //scenarios in current covering scenario group one by one
    {
        Pick up Group(i,n).GetAt(j) (the jth covering scenario in current covering group
        Combine Group(i,n).GetAt(j) with covering scenarios in covering scenario Group (i+1,n).
            If (an appropriate covering scenario is found)
            {
                Combine it with current picked covering scenario, and
                Insert the covering scenario after combination into appropriate scenario Group (I, n+1);
                Delete those two combined covering scenarios from their groups;
                        //Each covering scenario can only be combined once to guarantee disjoint
            }
            Else //Cannot find any covering scenario to combine
            {
                Store Group(i,n).GetAt(j) into final covering scenario set   //It is a final covering scenario
                Delete Group(i,n).GetAt(j) from Group (I,n);
            }
    }
  }
}
```

3.4 Experimentation and Scalability

The proposed algorithm has been implemented in a WS test case generation tool. To evaluate the scalability of the tool, we applied it to different types of systems with different sizes in different domains including WS, command-and-control application, semiconductor manufacturing application, and communication processors. The types of the applications include centralized, distributed, real-time, and embedded systems; and the sizes range from small experimental systems to large complex industrial systems.

A selected sample set of experimental results is shown in Table 1. The first two systems are experimental systems, and the other four systems are large industrial applications. The ICS (Industrial Control System) is the most complicated project with 140 scenarios, each of which is large and complex with thousands of pages of documents to describe the scenario. Writing the specification of the scenarios alone took more than 6 months to complete! RCS and LDRS are components of a command-and-control system. Table 1 also shows the time needed for the tool to complete the analysis. The results show that the tool works not only for small projects like the Car Alarm system, but also for large industrial projects such as ICS. For the ICS project, it took the tool only 961 seconds or 16 minutes to complete run on a PC. As can be seen from the table, the larger applications often have a larger number of incomplete CEs.

3.5 Swiss Cheese (SC) Approach

According to the WS testing process in Figure 1, after positive and negative test script sets are generated using pattern-oriented test script generation approach, the following step is to use Swiss Cheese (SC) approach to select the most error-prone and safety-critical test cases to save the testing cost [11][13]. SC approach takes WS specifications (in the form of Boolean expressions) as input and selects most critical test cases according to two criteria: *Hamming distance* (HD) and *boundary count* (BC) [11][12][13]. Both criteria are defined on the extended multi-Dimensional Karnaugh-map, which is called Swiss Cheese map. An example Swiss Cheese map is shown in Figure 3.

Both criteria HD and BC are proved to reflect the fault detection potency, therefore we use the pair (HD, BC) to evaluate the criticality of a test case (corresponding to a cell in SC map) in terms of fault detection potency. We defines a series of selection

criteria based on the (HD, BC) pair [13]. Generally speaking, a test case is more error-prone if the absolute value of its HD is smaller, but its BC is larger.

Table 1. Experimental Results Using the Tool

System	# of scenarios	# of conds	# of missing CE Comb.	# of partitions	# of Missing CE comb. after Partition	# of covering scenarios	Time (Secs)
Car Alarm Systems	13	12	547	1	547	40	127
Banking System	23	2	0	2	0	0	19
ICS	140	15	396,800	15	2,510	29	961
RCS	54	10	1792	3	18	4	149
LDRS	10	19	1,966,080	4	9,224	8	323
Communication Processor	31	33	$1.29 * 10^{11}$	29	80	27	329
Manufacturing Process Control	26	56	$1.35 * 10^{16}$	21	2	2	146

Table 2. Original specification of Flood Watching WS

Systems	Index	Specification
WS Server specification	1	Upon receiving a request from a client WS for a ranked water level list, the server WS sends the ranked water level list during the same period in the past 10 years back to the requesting client if the client's address is available and the database (DB) is not empty.
	2	Upon receiving a request from a client to query current water level, the server WS sends current water level to the requesting client if the client's address is available and current water level exists in the DB.
Client Side Specification	3	After a client sends a request for a ranked water level list, it waits for the server WS to send back the list. The client should be able to view the ranked water level list on client side.
	4	After a client sends a request for a ranked water level list, it waits for the server WS to send back the price. The client should be able to view current water level on client side.
Integrated Specification	5	There are multiple client WS. Only one of them sends a request for a ranked water level list at the same time. If the client's address is available and the database (DB) is not empty, the server WS sends the ranked water level list to the requesting client. The client should be able to view the ranked water level list on client side.
	6	There are multiple client WS. Only one of them sends a request for current water level at the same time. If the client's address is available and current water level exists in the DB, the server WS sends the water level to the requesting client. The client should be able to view the water level on client side

Table 3. The results of C&C analysis on the original specification

Specification	# of Scenarios	# of Conditions	# of Missing C-E Combinations	# of Covered Scenarios	Whole/ Partial
WS Server specification	2	3	12	4	Whole
Client Side Specification	2	3	12	4	Whole
Integrated Specification	2	7	248	10	Whole

Table 4. The results of C&C analysis on the refined specification

Specification	# of Scenarios	# of Conditions	# of don't care C-E Combinations	# of Covered Scenarios	Whole/ Partial
WS Server specification	2	3	8	2	Whole
Client Side Specification	2	3	8	2	Whole
Integrated Specification	2	7	176	6	Whole

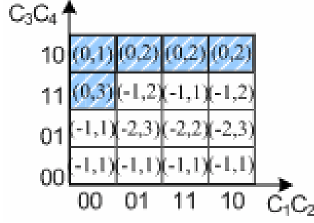

Figure 3. Swiss Cheese Diagram representing the test case set in Table 5

4. Case Study

This section uses a simple Flood Watching WS as an example to explain the WS development and testing process outlined above. In this example, the WS under development and test consist of a server WS and multiple client WS, residing in different locations. A client can send requests to the server and the server responses to the requests.

4.1 The original specification

Table 2 lists the original specification of the Flood Watching WS. The WS Server offers two functions and Client WS can access these two function.

4.2 C&C Analysis Results

This step is to perform C&C analysis on the original Flood Watching WS specification, repeatedly refines it until no uncovered conditions-event combinations exist. The statistical results of C&C analysis on the original Flood Watching WS is listed in Table 3. The number of missing condition-event (C-E) combinations represents how incomplete the specification is. The higher the number, the more incomplete the specification is. The last column explains whether current system is a whole system or only a part of it.

From Table 3 we can see that the original specification is incomplete. For example, the Integrated Specification has as many as 248 missing C-E combinations. All those missing C-E combinations need to be handled, either being supplemented or tagged as *don't care* C-E combinations, which are supposed not to affect system behaviors.

After repeatedly C&C analysis and specification refinement, we acquire a complete Flood Watching WS specification. Perform C&C analysis on it and we can get the following results: (listed in Table 4) From the table we can see that all missing C-E combinations have been supplemented based on C&C analysis results. For example, all the 72 missing C-E combinations are supplemented and only don't care C-E combinations are identified. The Flood Watching WS specification is complete now.

4.3 Swiss Cheese Model Experiment Results

Swiss Cheese Model has been studied in [11][12][13]. In this section, we illustrate how this model can be combined with C&C and verification patterns to generate test cases, using the Flood Watching WS example.

C&C analysis tool can automatically generate the Boolean expression corresponding to the 5th refined specification item in Table 2, which is:

$$S = \overline{C_1}\,\overline{C_2}C_3C_4 + C_3\overline{C_4}$$

This Boolean expression can be put into a Karnaugh-map styled SC diagram in Figure 3 where (HD, BC) pairs of each cell are tagged in the SC diagram.

The SC diagram can be represented in Table 5, where values in the cells are the HD, cell corresponds to a test case that check if the Boolean expression S is satisfied. Considering the topological hypercube structure of Boolean expressions, boundary count (BC) in the last column can be obtained [11][12][13].

SC approach selects the most safety-critical test cases and therefore saves the testing cost. For example, to test the 5th specification in Table 2, we need to run 16 test cases (5 positive and 11 negative) without SC model. But after ranking and selecting test cases with SC model, we only need to test the most critical test case set {0011, 0110, 1110, 1010, 0111, 1011}, and save (16-6)/16 = 62.5% testing cost. Further more, among all the 6 selected test cases, 4 positive test cases {0011, 0110, 1110, 1010} fall into the same pattern and the other 2 negative test cases {0111, 1011} fall into the same pattern. Pattern-oriented test case generation approach

[10] can further reduce testing effort from 25% to 90% depending on the on experience level of engineers involved and the kind of changes.

Table 5. Test case set generated for the refined 5th specification in Table 2

HD	Cells (test cases) C_1C_2 C_3 C_4	# of cells	BC
HD = 0	0011	1	3
	0110, 1110, 1010	3	2
	0010	1	1
HD = -1	0111, 1011	2	2
	0001, 1111, 0000, 0100, 1100, 1000	6	1
HD = -2	0101, 1001	2	3
	1101,	1	2
Total		16	

5. Conclusions

The major contributions of this paper are twofold. It defined an integrated process to check the completeness of conditions and generate test cases in OWL-S based WS specification and developed a more efficient algorithm to check the completeness. The C&C analysis can be used to generate both positive and negative test cases to verify the functionality and robustness of the WS under test. Even though the number of incompleteness identified can be rather large, the number of covering scenarios is often small, and thus makes this approach scalable to large Web applications. The overall process has also been automated and several large industrial applications have been experimented including several large real time distributed Web applications.

Reference

[1] D. Beyer, A. Chlipala, T. Henzinger, R. Jhala, and R. Majumdar, "Generating Tests from Counterexamples", Proceedings of the 26th International Conference on Software Engineering (ICSE'04), Scotland, UK, May 2004, pp. 326 – 335.

[2] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 2002.

[3] T. Y. Chen and M. F. Lau, "Test Cases Selection Strategies Based on Boolean Specifications", Software Testing, Verification and Reliability, Vol. 11, No. 3, Sep. 2001, pp. 165-180.

[4] J. Clune and L. Chen, "Testing Web Services: Methods for Ensuring Server and Client Reliability" at http://www.sys-con.com/websphere/

[5] N. Davidson, "Testing Web Services" at http://www.webservices.org/, in October 2002.

[6] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy Abstraction", In Proceedings of the 29th Annual Symposium on Principles of Programming Languages, 2002, pp. 58-70.

[7] G. Holtzman, "The Spin Model Checker," IEEE Transactions on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295.

[8] J. Myerson, "Testing for SOAP Interoperability" at http://www.webservicesarchitect.com/, Feb 2002.

[9] Wim De Pauw, et al., "Websight Visualizing the Execution of Web Services", Workshop on Testing, Analysis and Verification of Web Services, Boston, MA, July 2004.

[10] W. T. Tsai, R. Paul, L. Yu, and X. Wei, "Rapid Pattern-Oriented Scenario-Based Testing for Embedded Systems," in *Software Evolution with UML and XML*, edited by H. Yang, Idea Group Publishing, London, 2005, pp. 222-262

[11] W. T. Tsai, X. Wei, Y. Chen, B. Xiao, R. Paul, and H. Huang, "Developing and Assuring Trustworthy Web Services", ISADS, 2005, pp. 43-50.

[12] W. T. Tsai, Y. Chen, R. Paul, H. Huang, X. Zhou, X. Wei, "Adaptive Testing, Oracle Generation, and Test Case Ranking for Web Services", COMPSAC, 2005, pp 101-106.

[13] W. T. Tsai, X. Wei, Y. Chen, R. Paul and B. Xiao, "Swiss Cheese Test Case Generation for Web Service Testing", accepted by IEICE (the Institute of Electronics, Information and Communication Engineers)/IEEE Joint Special Section on Autonomous Decentralized Systems, 2005.

[14] K. H. Rosen, Discrete Mathematics and Its Applications, 5th edition, McGraw Hill, 2003.

[15] T. Ninomiya, and M. Mukaidono, "Independence of the Axioms of Boolean Algebra in Multiple-Valued Logic", Multiple-Valued Logic, 2000. (ISMVL 2000) Proceedings. 30th IEEE International Symposium on , 2000, pp. 107 – 112.

[16] T. Ninomiya, and M. Mukaidono, "Independence of Each Axiom in a Set of Axioms and Complete Sets of Axioms of Boolean Algebra", Multiple-Valued Logic, 2002. ISMVL 2002, Proceedings 32nd IEEE International Symposium on Multiple-Valued Logic, 2002, pp. 185 – 191.

[17] B. C. H. Turton, "Extending Quine-McCluskey for Exclusive-Or Logic Synthesis"; IEEE Transactions on Education, Vol. 39, No. 1, Feb. 1996, pp. 81 – 85.

[18] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated Consistency Checking of Requirements Specifications", ACM Teansactions on Software Engineering and Methodology, Vol. 5, No. 3, July 1996, pp. 231-261.

[19] F. Mileto, G. Putzolu, "Statistical Complexity of Algorithms for Boolean Function Minimization", Journal of the ACM (JACM), Vol. 12, No. 3, July 1965, pp. 364-375.