Testing Web Services Using Progressive Group Testing

Wei-Tek Tsai[1], Yinong Chen[1], Zhibin Cao[1], Xiaoying Bai[2], Hai Huang[1], and Ray Paul[3]

[1] Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287-8809, USA
[2] Tsinghua University, Beijing, China
[3] Department of Defence, Washington DC, USA

Abstract. This paper proposes progressive group testing techniques to test large number of Web services (WS) available on Internet. At the unit testing level, the WS with the same functionality are tested in group using progressively increasing number of test cases. A small number of WS that scored best will be integrated into the real environment for operational testing. At the integration testing level, many composite services will be constructed and tested by group integration testing. The results of group testing at both unit and integration levels are verified by weighted majority voting mechanisms. The weights are based on the reliability history of the WS under test. A case study is designed and implemented, where the dependency among the test cases in WS is analyzed and used to generate progressive layers of test cases.

Keywords: Web Services, Service-Oriented Architecture, Verification and Validation, Web testing.

1 Introduction

Numerous approaches are possible to generate unit testing for Web services (WS) [4], e.g., generate test scripts based on functional description, based on process description (such as the process description proposed by DAML-S [11], OWL-S Web Ontology Language or the ACDATE scenario model [8], based on various coverage criteria (object methods, sequence, decision and condition coverage), or based on sequence constraints such as MtSS and MgSS [1] The issue for WS is that in most cases, only the service providers have the full access to the WS white-box specification, while service brokers and service clients have access to WS description only such as WSDL or OWL-S description. In [5], it was suggested that WSDL should include some of these information so that test cases/scripts can be automatically generated. The group testing scheme that we proposed in [1] is powerful and efficient, yet its improvement over the individual sequential testing is a linear function of the number available computers. It may not cope with the possible exponential growth of WS available over the internet. This paper presents a modified and more aggressive group testing scheme to combat this problem.

C.-H. Chi and K.-Y. Lam (Eds.): AWCC 2004, LNCS 3309, pp. 314–322, 2004.
© Springer-Verlag Berlin Heidelberg 2004

In [2], Bloomberg suggested to develop WS testing technologies in three phases. In phase one, WS are mainly tested like the ordinary software. In phase two (2003-2005), the following features should be included in testing: WS, the publishing, finding, and binding capabilities, the asynchronous capabilities of WS, the SOAP intermediary capability, and the Quality of services. In phase three (2004 and beyond), dynamic runtime capabilities, WS orchestration testing and WS versioning should be tested.

WS-I (Web Services Interoperability Organization), an industry organization chartered to promote WS interoperability across platforms, released a WS testing tool in March 2004 [http://www.ws-i.org]. The tool consists of two components: WS-I monitor and WS-I analyzer. The WS-I monitor can be placed between the client and the WS. It logs all messages, requests and responses, as they go back and forth. The WS-I analyzer then goes through every message in the log and analyzes them against all the interoperability requirements.

In the past few years, we developed in the past few years techniques related to testing and verification of WS. In [6], a variety of test generation techniques are proposed to test WS in an enhanced UDDI-based service broker. These test scripts can be arranged hierarchically to test domains of related WS. An early form of WS check-in and check-out processes are also proposed to increase the confidence of WS used. In [1, 7, 9], rapid testing techniques were developed, including group testing, regression testing, and pattern-based verification. In [7], WS scenarios are developed in stages, the first stage the individual service scenarios are developed, in the second stage, interaction among WS are modeled, and finally the overall scenarios are developed combining previously developed scenarios. These scenarios are then translated into test scripts to be performed by organized distributed agents. In [5], an enhanced WSDL interface was developed to include dependency information, functional description, invoking, and concurrent sequence specifications so that test cases/scripts can be generated based on WSDL descriptions.

This paper proposes progressive group testing techniques to test possible large number of WS available on online, at both unit testing and integration testing levels. The sequence constraint concept is used to generate the progressive sets of test cases. The rest of the paper is organized as follows. Section 2 outlines the trustworthy computing model on WS. Section presents group testing for at unit test level and section 4 extend the testing scheme to integration test. Section 5 study the description of WS, the test case generation, and the test case partitioning based on their dependencies. Section 6 concludes the paper.

2 Progressive Group Unit Testing

WS-based computing has an open platform that allows service providers and requestors freely adding and accessing the available services. As a result, huge amount of check-in, check-out, and acceptance testing need to be performed. Progressive group unit testing and group integration testing are designed to address this problem. Group testing technique was originally developed for testing

large samples of blood [3] and this paper uses it to test large number of WS submitted to the trustworthy service provider at runtime. It tests the “contamination” of an entire group of services by applying one test. The principles of this new testing scheme include:

- The platform is open and everyone can submit WS for consideration;
- Testing must be completed within available time frame, possibly to meet a real-time deadline;
- WS are evaluated and competed based on objective measures;
- Only the best WS will be accepted.

2.1 Prescreening

At the beginning, fast testing process is applied to immediately eliminate the unlikely-to-win candidates. The sophistication of testing increases progressively and the final winners will be tested rigorously. The system will pick up multiple winners with ranking. The reasons to accept multiple services with the same functionality are (1) Fault tolerance: When a service is unavailable, a backup spare can replace it; (2) Parallel processing: When the demand is high, multiple services will be used to process the same kinds of requests; (3) The ranks of services could be linked to the price. To implement this scheme, we organize our rigorous unit testing scripts into a hierarchy according to the dependency relationship among test cases. We define a dependency relationship R on a set of test cases T as a binary tuple (u, v) which means that test case u fails implies that test case v will fail. Examples for such dependency will be provided later for both unit testing and integration testing. Assume no cyclic dependency, the graph of R on T is a directed acyclic graph (dag) and a topological sorting will define the hierarchy (a leveled graph). The hierarchy may save tremendous cost on testing since when a WS fails a test case u, we know for sure that the WS will fail all test case v such that (u, v) R. Hence we will apply test cases on a WS in the order defined by the hierarchy. Multiple strategies are feasible to rule out a WS:

1. Based on the number of test cases a WS fails in one level of the hierarchy
2. Based on the criticality of test cases a WS fails in one level of the hierarchy
3. Based on the weight of test cases a WS fails in one level of the hierarchy.

For each strategy, there are different levels of tolerance. For example, in strategy 1, the strictest tolerance is that if a WS fails one test case, it will be dropped. The least strict tolerance is that the WS fails all test cases. We can also have a predefined threshold M so that we only drop a WS if it fails more than M test cases. The level of each strategy is depicted in the following table. One important factor that affects the tolerance is the competitiveness. If many service vendors provide a same WS, then the tolerance should be strict and vice versa.

We would like to remark that a WS fails M test cases in one level does not necessarily imply that it will fail more than or equal to M test cases in the next level. This is because test cases may have different number of descendants. For a

	Strictest	Moderate	Least Strict
# of test cases	1 test case fail	# of failures is greater than M	All fail
Criticality	1 critical test case fail	# of critical test case failures is greater than M	All critical fail
Weight	positive weight fail	# of weight failures is greater than M	Total weight fail

test case u, define its descendants as a set of test cases D(u) such that v D(u) if (u, v) R or there exists a test case w, such that (w, v) R. A good weight function for test cases could be the number (total weight) of test cases in its descendants. The number of test cases applied on a WS increases as it passes more levels in the hierarchy as shown in Figure 1. The rank of a WS is defined as the number (or weight) of test cases it fails so far. The rank serves as an additional criterion to rule out extra WS. The basic algorithm is as follows: The "not applying descendants test cases" in step 4 will further reduce the cost of the testing, in addition to the ruling out of WS at each level. In the prescreening process, voting mechanism can be used to determine if an atomic WS has performed the desired service.

1. Identify dependency among test cases
2. Do topological sorting on test cases and form the hierarchy
3. Starting from level 1 in the hierarchy, for each level
4. Apply all test cases in the current level on all survivors WS, except those in the descendants of some test case u that the WS fails before.
5. Rule out some WS
6. Rule out extra WS according to their ranks if too many WS survive
7. end

2.2 Runtime Group Testing

The best candidates identified in prescreening step will be integrated into the live system and further tested in runtime.

Figure 2 illustrates the part of the system in the trustworthy service broker that performs runtime group testing. Assume CSn is a composite service consisting of n services S1, S2, ..., Sn, where Si can be an atomic service or a composite service. While CSn is performing services, many atomic services could be registered. Assume services S11, S12, ..., S1m are functionally equivalent to the service S1 in SCn. We can forward (broadcast) the input to S1 to S11, S12, ..., S1m, as shown in Figure 2. The results from all services, including that from S1, are voted by a voting service. The voting is weighted based on the current reliabilities of the services under test. The voting service can set the initial weight of each incoming service to zero while the exiting service S1's weight to the reliability R(S1). The voting service detects faults by comparing the output of each service with the weighted majority output. A disagreement indicates a fault.

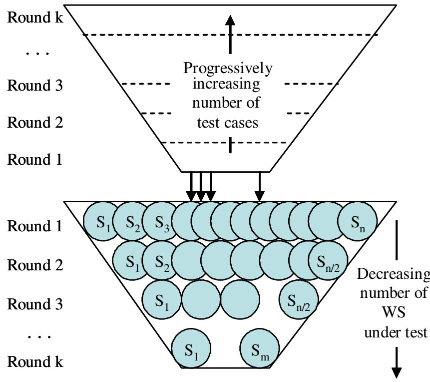

Fig. 1. Progressive group testing

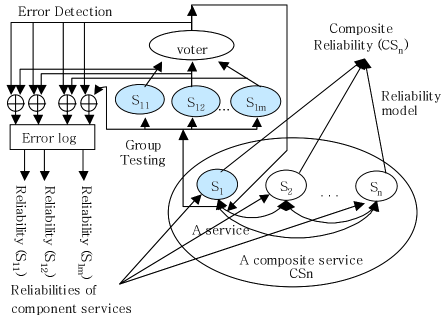

Fig. 2. Group testing scheme

Based on the failure log generated by group testing, the reliabilities individual services under test can be evaluated [10].

The novelty and features of the group integration testing include:

– One of the tough problems in software testing is to construct an oracle that can determine if a failure has occurred. In this group integration testing scheme, the voting service serves as the oracle according to the majority principle.
– Another advantage is that testing is done while performing the normal operations. In other words, the WS under test are actually integrated into the real operational environment without a separate testing phase at no extra time, provided that sufficient computing power is available. A remote test agent can take spare computers on the Internet to perform distributed testing was proposed to address this problem [1].

Both prescreening and runtime testing applied the group testing principle. The major difference is the test cases applied. The former applied the progressive test cases designed for testing while the latter apply the runtime environment to drive the unit under test.

There is a possibility that the majority voting may fail to choose the best service, if a malicious service provider submit thousand of the WS to gain the majority. This potential problem is addressed as follows:

– All service provider must register and be certified. The submission of a WS must append the digital signature of the service provider.
– The voting is weighted. The WS that have a known oracle or a track record of producing reliable output will be given higher weights. The newly submitted WS will be assigned an initial weight of zero, which mean it's output will not be used to form the majority output. When the output of a WS agrees with the majority, its weight is increased.
– The reliability of each service is stored in a database together with the profile of the service.

– For some test inputs, it is relatively easy to obtain the expected outputs. During the prescreening or group testing (in either group unit testing or group integration testing), run those test cases first. This will identify some of the malicious code.

If a component WS, e.g., S1, has not gone through unit testing, the unit testing voter can check their outputs to make sure they are correct. The outputs of the composite services are sent to the integration testing voter for error detection.

3 A Case Study

This section defines two banking systems to illustrate the service compositions and test case generation for progressive unit and integration testing, respectively.

3.1 Atomic Service and Unit Testing

Before a WS and a composite WS are integrated in the actually system, we need to generate test cases to drive testing and we need to divide the test cases into layers for progressive testing. We first define a simplified bank system that offers a single service and use this service as an example for unit test. The functions of the service are listed Table 1. As unit test, we designed 35 test cases, named TC1, TC2, ..., TC35, to test the functions in Table 1.

Table 1. The functions of a banking service

Service Name	Function Description
Login	Provide authentication function for the ATM system. Many other operations are based on user's correct login.
Check Balance	Check the balance and return the result to the client.
Update Account	Update the account information in the database
Check Account	Check if the account ID or other information exits in the database
Delete Account	Delete the account and process other necessary functions, such as withdraw all remaining money.
Create Account	Create a new account
Withdraw Money	Withdraw money from the account. Require login correctly
Deposit Money	Deposit the money into account. Request to login correctly
Get Account Information	Get all information for an account, such as the balance and interest

In order to perform progressive testing, we need to divide the test cases into progressive layers. We divide the tests cases based on the dependencies among the functions. For example, WithdrawMoney function in Table 1 depends on the success of Login function. Based on the dependencies among the functions, we can obtain the dependencies among the test cases as follows: A test case

TCj depends on TCi, if the failure of the test applying TCi will lead to the failure of the test applying TCj. The dependencies among the functions must be assigned manually while the dependencies among the test cases can be computed automatically based on the functions dependencies.

Figure 3 shows the automatically generated dependency diagram among the 35 test cases. Once we obtained the dependency diagram, we can automatically obtain the progressive layer of each test case. Table 2 divides the 35 test cases into four layers for progressive testing, i.e., layers 1, 2, 3, and 4 will be tested in four different rounds.

Table 2. Layers of test cases for unit test

Layer	Test Case Number
1	TC1, TC2, TC3, TC4, TC5, TC6, TC7, TC8, TC9
2	TC10, TC11, TC14, TC15, TC16, TC17 TC18, TC19, TC20, TC21, TC22
3	TC12, TC13, TC32, TC33, TC34, TC35
4	TC23, TC24, TC25, TC26, TC27, TC28, TC29, TC30

Table 3. Layers of test cases for integration test

Layer	Test Case Number
1	TC1, TC4, TC11
2	TC2, TC8, TC10
3	TC6, TC9, TC7, TC3
4	TC5, TC12, TC13

For testing the integrated service, we use MfSS specification [?] to describe the test cases. Each test case use at least two services' functions. These functions are the integrations of different service.

In this example, there is some functional dependence which must be specified manually. And the ranking of all other test cases will be effected. For example: GetAccountInformation gets the information related to an account, including the current balance information which is also provided by the function CheckBalance. The function CreateAccount depends on the function CheckAccountID, since if CheckAccountID failed, system can not process CreateAccount. From the MfSS specification of TC1 and TC2, we can see that the only difference between TC1 and TC2 is that TC1 uses CheckAccountID and TC2 uses the

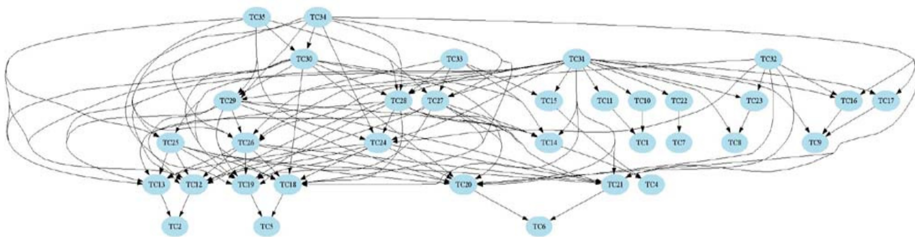

Fig. 3. Dependencies among test cases

CreateAccount. Thus in our ranking algorithm, TC1 belongs to the first layer, and TC2 is the second layer. According the dependency relationship, we can define the layers of test cases for progressive testing. Table 3 divides the 13 test cases into four layers.

3.2 Composite Service and Integration Testing

Then we define more complex system with three services that offer different subfunctions. Tables 4 and 5 show the two composite services examples, respectively. The following two composite services are used to form another larger composite service: the banking service. They are used to design the integration test cases in this case study.

Table 4. WS1: ATM Service

Service Name	Code	Service Description
Check Balance	m11	Check the balance for the user and return the number to the client.
Withdraw Money	m12	Withdraw the money from the account. Request to login correctly
Deposit Money	m13	Deposit the money into account. Request to login correctly
Get Account Information	m14	Get all information for an account, such as the balance and interest

Table 5. WS2: Authentication Service

Service Name	Code	Service Description
Login	m21	Provide the authentication function for the ATM system. Many other operations need user to login correctly at the first.
Update Account	m22	Update the account information in the database.
Check Account	m23	Check if the account ID or other information is not already used in the database.
Delete Account	m24	Delete the account and process other necessary functions, such as withdraw all left money.
Create Account	m25	Create a new account.

4 Conclusion

We proposed progressive group testing techniques for unit and integration testing of WS. The techniques are designed to test large number of WS available on the Internet rigorously and at runtime. The major contributions of the paper are threefold: Progressive test case generation, group testing with majority voting, and weighting the reliability history of WS. The definition and the efficient selection algorithm of progressive test cases ensure that WS are tested by applying an optimal sequence of test cases. Group testing ensure that WS are

test in parallel and their results are verified by weighted majority voting. The WS that have proven history of reliable behavior will have a higher weight while the new WS will start their initial weight from zero, which means they have no influence to the majority output at the beginning. The weights of WS increase dynamically when their outputs agree with the majority outputs.

References

1. X. Bai, W. T. Tsai, T. Shen, B. Li, and R. Paul, “Distributed End-to-End Testing Management”, EDOC 2001: 140-151
2. J. Bloomberg, “Web Services Testing: Beyond SOAP”, ZapThink LLC, Sep 2002, http://www.zapthink.com
3. D. Z. Du and F. Hwang, “Combinatorial Group Testing And Its Applications”, World Scientific, 2nd edition, 2000
4. A. D. Gordon and R. Pucella, “Validating a Web Service Security Abstraction by Typing”, Microsoft MSR-TR-2002-108, December 2002
5. W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang, “Extending WSDL to Facilitate Web Services Testing”, Proc. of IEEE HASE, 2002, pp. 171-172
6. W. T. Tsai, R. Paul, Z. Cao, L. Yu, A. Saimi, and B. Xiao, “Verification of Web Services Using an Enhanced UDDI Server”, Proc. of IEEE WORDS, 2003, pp. 131-138
7. W. T. Tsai, Lian Yu, Feng Zhu, and R. Paul, “Rapid Verification of Embedded Systems Using Patterns”, Proc. of IEEE COMPSAC 2003, pp. 466-471
8. W. T. Tsai, R. Paul, L. Yu, A. Saimi, and Z. Cao, “Scenario-Based Web Service Testing with Distributed Agents”, IEICE Transactions on Information and Systems, 2003, Vol. E86-D, No. 10, 2003, pp. 2130-2144
9. W.T. Tsai, N. Liao, H. Huang, and R. Paul, “Application of Group Testing in Verification of Dynamic Composite Web Services”, in Workshop on Quality Assurance and Testing of Web-Based Applications, in conjunction with COMPSAC, September 2004, pp. 28-30
10. W. T. Tsai, D. Zhang, Y. Chen, H. Huang, Ray Paul, and N. Liao, “A Software Reliability Model for Web Services”, submitted to the 8th IASTED International Conference on software engineering and applications, Cambridge, MA, November, 2004
11. D. Wu, et al., Automating DAML-S Web Services Composition Using SHOP2, http://www.mindswap.org/papers/ISWC03-SHOP2.pdf