Ontology for Service Oriented Testing of Web Services

Yufeng Zhang
Dept of Computer Sci, National Univ. of Defence Tech., Changsha, China, Email: yuffonzhang@163.com

Hong Zhu
Department of Computing, Oxford Brookes University, Oxford OX33 1HX, UK, Email: hzhu@brookes.ac.uk

Abstract

This paper presents a service oriented architecture for testing Web Services. In this architecture, various parties interoperate with each other to complete testing tasks through testing service registration, discovery and invocation. The analysis of the architecture in a typical scenario shows that it has the advantages of supporting dynamic discovery and invocation of testing services as required by the dynamic discovery and invocation of normal functional services without compromising security, privacy and intellectual property rights. It is flexible and extendable. It also helps to reduce the risk of unnecessary disturbances to the normal operations of services due to testing activities. The paper reports a prototype implementation of the architecture by adapting and implementing the ontology of software testing using Semantic Web Services technology. A case study with the WS wrapping of an automated testing tool is also reported, which demonstrated that the architecture is technically feasible.

1. Introduction

The recent development of web technology marks the beginning of a new era of service oriented computing. In particular, Web Services (WS) enable applications to communicate with each other over the Internet [1]. Semantic Web facilitates the definition of the semantics of information and services on the web, making it possible for the web to understand and satisfy the requests of people and machines to use the web content [3]. The combination of these two, i.e. Semantic Web Services (SWS), uses the Semantic Web to help to create a repository of computer readable data and to describe the semantics of the services that perform tasks and transactions. It supports capability-based service discovery and interoperation at runtime. This opens up a huge range of new applications and a new platform of great flexibility.

However, quality assurance and testing of WS applications is still an open problem. On one hand, loose coupling of services improves system testability. However, on the other hand, the difficulty of testing WS applications increased due to the poor observability and controllability [4]. Services are independent entities that control their own resources and behaviors autonomously and collaborate with each other actively and automatically [2]. Their autonomous and dynamic behaviours make the observation of test results and the control of testing process much more difficult, if not impossible.

In the literature, research efforts on quality assurance and testing of WS applications have been reported. Chan and Cheung applied metamorphic testing methods to test WS and treat WS as black box [5]. Tsai and Paul proposed to extend the WSDL to support WS testing by providing additional information in WS description [6]. In [7], network level fault injection was used to test WS applications. Some other methods have also been proposed, such as using data perturbation to generate test cases for WS [8] and testing the semantics of XML Schema [9], etc. However, the difficulties in testing WS caused by the autonomous nature of WS and the need of testing on-the-fly are still not addressed.

To meet these challenges, in [4] Zhu proposed a service oriented framework. In this framework, various parties interoperate with each other to perform testing tasks via test service search, invocation and delivery. He also proposed the utilization of an ontology STOWS of software testing to enable the collaboration between testing services. This paper is based on the framework. It further develops the techniques by implementing it in Semantic Web Services and demonstrates its feasibility.

The remainder of this paper is organized as follows. Section 2 outlines the architecture and analyzes the workflow of testing tasks to show how the services collaborate with each other. Section 3 presents the ontology. Section 4 describes the implementation of the framework using techniques in Semantic Web Services. Section 5 reports a case study to show how a testing tool is wrapped into SWS. Section 6 concludes paper with a remark on future work.

2. SOA for Testing WS

This section first outlines the architecture and then analyzes a typical scenario in testing a web service.

2.1 Overview of the architecture

Figure 1 shows the overall structure of the architecture proposed in [4]. In this architecture, a WS should ideally be accompanied with a specially designed service that facilitates the online testing of the original services. For the sake of convenience, the original services that provide functions for costumers are called *functional services* (or shortly *F-services* in the sequel). The special services that designed to help testing the functional services are called *testing services* (*T-services* for short), which are provided either by the functional services providers or a third party.

978-0-7695-3499-2/08 $25.00 © 2008 IEEE
DOI 10.1109/SOSE.2008.35

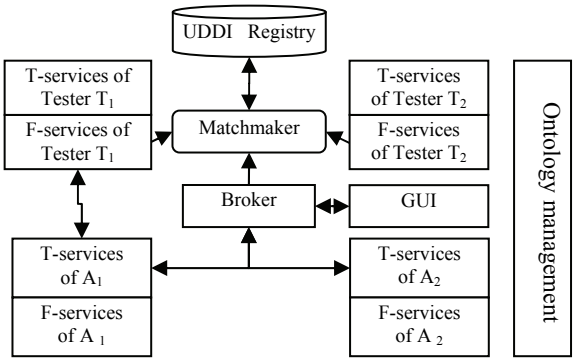

Figure 1. System Architecture

In Figure 1, A_1 is the service that to be tested. The T-services of A_1 (i.e. *testing services*) is designed specially for testing the F-services of A_1. *Testers* are third party service providers specialized in providing software testing services, such as testing tool vendors. These testers can perform general testing tasks such as test case generation, test execution, measuring test adequacy and so forth. They could have their own *T-services* that enable testing themselves. Testing tasks are performed by the collaboration of these loose-coupled testing services.

It is worth noting that the general service oriented architecture is insufficient to achieve the purpose of online testing of WS, because testing tasks are usually too complicated to be performed by one testing service and need dynamic generation of test plan and execution of the test plan through the collaboration of multiple testing services. This problem becomes apparent in the analysis of a typical scenario in the next subsection. A solution to this problem is to introduce the notion of testing service broker, which is a special type of testing services that coordinates the testing services to ensure test tasks performed correctly. It receives the requests of testing tasks from test requesters, makes test plans, decomposes the test plans into subtasks, searches for and invokes other testing services that are capable of performing the corresponding tasks according to the plan. Search for appropriate testing services is another difficulty. Our technical solution is to use a Matchmaker to collaborate with UDDI to provide testing service registration and search facility. It is a searching engine of testing services registered in UDDI.

For this idea to be practically workable, some technical issues must be addressed. First, an effective communication mechanism for these WS is needed. Entities involved in this framework are loose-coupled WS. The bindings of services may happen at runtime. This requires that the artefacts should be encoded in machine readable standard code so that services can understand them correctly. Second, the services should be searchable according to their capabilities. These issues can be achieved using Semantic Web Services techniques.

2.2 A typical scenario

In order to illustrate how the proposed architecture works and to identify the technical issues in the implementation, let's analyze a fictional typical scenario.

Suppose that a bank is developing or running a WS called *FM* to serve its team of fund managers to buy and sell shares through stock market brokers and to serve its customers to buy and sell fund online. In order to connect to a WS provided by a stock market broker, say *SB*, it is required to test *SB*'s WS with adequate combinations of parameters. Note that the connection to *SB* could take place at runtime as a result of searching the UDDI. In our proposed architecture, *SB* should provide a *testing service* (*T-services*) in order to separate testing transactions from the real transactions. The later will require real monies change hands and share account state updates; while the former does not.

The process of testing starts with the request of a testing task by *FM* either manually or automatically. The task should consist of a test objective to be achieved and the target service to be tested. For example, the test objective might be to check every equation in a formal algebraic specification of the service provided by *SB*.

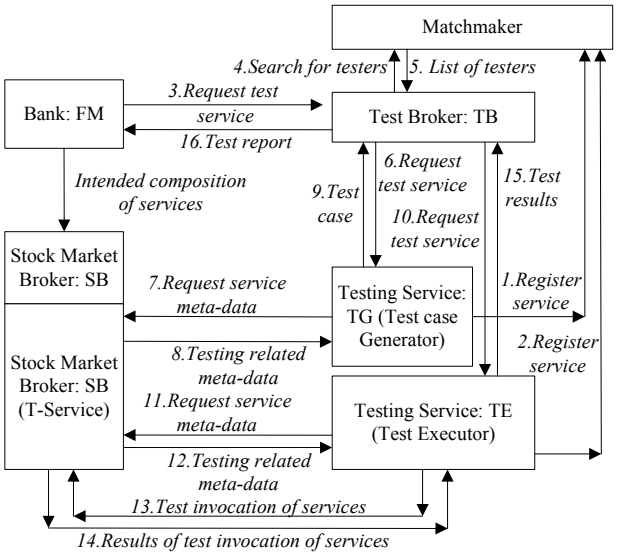

Figure 2. The scenario of testing banker-broker composition

This test task is represented in a message and submitted as service request to a test broker *TB*. After receiving the request submitted by *FM*, *TB* makes a test plan according to the test requirement and decomposes it into a sequence of smaller test subtasks if necessary. In this particular scenario, suppose that *TB* decomposes the test task into two subtasks: (a) the generation of test cases, and (b) the execution of the test cases and checking the correctness of test results.

In order to get these two testing tasks performed, the test broker *TB* searches the UDDI registry for each subtask through a matchmaker. In this particular scenario, *TB* will search for a test case generator and a test execu-

tor and oracle. There may be multiple registered WS that are capable of performing a certain testing task. One of these candidates must be selected by the requester according to certain criterion, such as its quality of services. We suppose that *TB* selects *TG* for generating test cases and *TE* for executing the test cases and checking the correctness.

In the next step, the test broker constructs a sequence of requests and submits them to the corresponding selected test services. For example, *TB* sends a request of generating test cases to *TG* with the information about the artefacts involved in the testing, such as an algebraic specification of *SB*.

A testing service may perform a test task solely based on the information contained in the test request, or contact the related WS to obtain necessary information. For example, *TG* may interact with *SB*'s *T-service* for the information about its source code and the metadata about the source code such as its language.

It is worth noting that first we assume that *TG* can be trusted by the service provider *SB*. A mechanism can be set up for certifying the legitimacy of third party testing service providers and agreeing on proper dealing with information privacy and intellectual property rights. Second, some testing tasks may need human participation. Thus, a testing process can be a long transaction and WS can serve as a human computer interface. In the sequel, we will not distinguish manual realization of a service from automatic realization as far as the messages passing between roles are standard and machine readable.

Once *TG* completed the task of test case generation, it sends the generated test cases to *TB*. *TB* will then send test cases together with other related information to *TE* to make a service request. *TE* will then invoke the test executions of *SB* and check the test results. Once finished the testing task, it will return to *TB* with a test report.

Generally speaking, an invocation of a service as a test should be submitted to the testing services so that it can be distinguished from a real request of the services. Otherwise, an invocation of a service must carry a tag to signal whether the service request is a test. Note that, in the WS standard stack, there is no mechanism that supports the distinction of normal service requests from testing requests. Moreover, there are other testing related services that are necessary to enable automated on-the-fly testing of WS. For example, in this particular scenario, we need testing services to grant permissions to access the source code, formal specification and/or other metadata of the services, to report the test coverage of test executions, etc. Here, we assume again that testers *TE* and *TG* are trusted by *SB*. Checking if a tester is legitimate is also an important function of testing services. Therefore, in general, to separate testing services from the functional services is a reasonable design decision.

As illustrated by the above scenario, this collaboration among multiple roles consists of service search, service invocation and service execution. In this process, *FM* eventually achieves its test objectives while the service provider *SB* does not lose its intellectual property rights because the sensitive information is only released to trusted third party specialized in testing.

2.3. Analysis of the scenario

From the above illustrative scenario, we can identify the following key technical issues of the interaction process.

(a) *How to describe the capability of a testing service*?

Testing services must be searchable according to their capabilities so that they can be discovered at runtime. The matching between search request and service registry is the key to the successful discovery of services.

(b) *How to invoke testing services*?

Invoking a service at runtime may involve a number of software artefacts, such as the program/service under test, the test cases, the specification of the service, the execution results, etc. The interaction between the service provider and the service requester may also be a complicated process.

These issues can be achieved by using the Semantic Web Services (SWS) technology, in which the concepts in the topic domain of software testing, such as tasks, capabilities, test methods and artefacts can be defined in the form of ontology. Testing service registration, requests and their results are also represented using the terminology defined by the ontology. The following will present such an ontology and its implementation in OWL.

3. Ontology of Software Testing

Generally, ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for the combination and extension of the vocabulary [10]. It articulates a domain specific knowledge [11].

The Web Ontology Language OWL is a semantic markup language for publishing and sharing ontologies on the Web [12]. It is designed for applications that need to process the content instead of just presenting information to humans [13].

We adapt the ontology of software testing STOWS built in [4, 11], which was originally developed for agent oriented software testing. Its concrete representation of many concepts does not fit well into the architecture of service oriented computing and the OWL-S standard.

The revised ontology includes basic concepts *Tester*, *Activity*, *Artefact*, *Context*, *Environment* and *Method.* They are combined together to express compound concepts *Capability, and Task,* which can be represented in OWL-S Service Profile. The following describes each concept one by one.

(1) *Tester*. A tester refers to a particular party who carries out a testing activity. In general, testers include hu-

man beings, organizations and software systems. In the context of service orientation, we only consider testing services as testers. All the testing tasks are performed by services. A tester can therefore be an atomic test service, or a composition of testing services. An important attribute of a *Tester* is its capability, which will be discussed later.

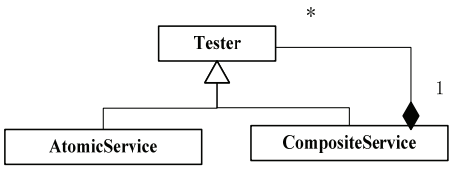

Figure 3. Tester

(2) *Activity*. There are various testing activities including *test planning*, *test case generation*, *test execution*, *result validation*, *adequacy measurement* and *test report generation*, etc. [11].

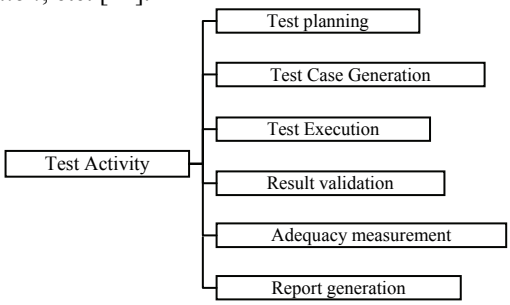

Figure 4. Test activities

(3) *Artefact*. A test task performed by a service may involve multiple kinds of artefacts. The *Artefact* possesses an attribute *Location* expressed by a URL or a URI to give the location of the artefact on the Internet.

(4) *Method*. For each test activity, there may be multiple testing methods applicable. Method is a part of the capability and also an optional part of test task. Test methods can be classified in a number of different ways. Figure 5 show two typical classifications of the concept *Method*. Both of them are represented in the hierarchy of test methods in the ontology.

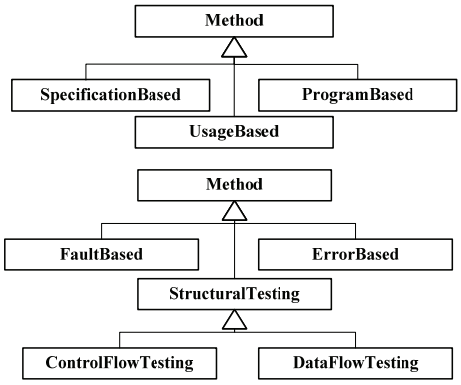

Figure 5. Concept of Method (b)

(5) *Context*. Testing activities may occur in different software development stages and have various testing purposes. Testing contexts typically include unit testing, integration testing, system testing, regression testing, etc. (6) *Environment*. The testing environment is the hardware and software configurations in which a testing is to be performed.

These concepts in the ontology are managed by the ontology management module in the framework. Details are omitted for the sake of space.

4. Implementation

This section describes how the key technical aspects are implemented using SWS technology.

4.1 Description of capability and task

Generally speaking, there are two basic capability representation approaches for WS [16, 17]. The first is based on hierarchical classification of services in which each class represents a set of services capable of performing the similar task (i.e., of the similar capability). The second implicitly describes the capability of a service by its state transformation and the information transfers. OWL-S combines these two and uses ontology of services. It describes services in three main parts: Service Profile, Service Model and Service Grounding [18]. Service Profile represents the capability of a service by describing its category and IOPE (Input, Output, Precondition and Effects) [18]. The registration of and search for a service are all based on the Service Profile.

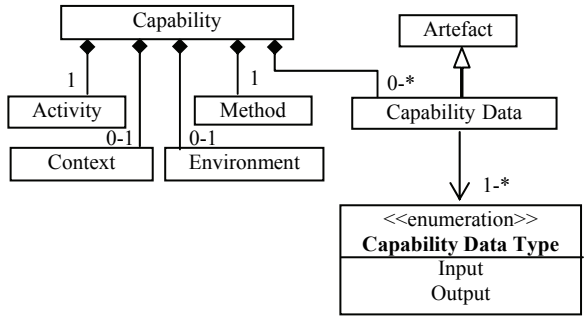

Figure 6. Structure of capability in the ontology of [11]

Conceptually, the search of a test service is to match service's capability with the required test task. In the STOWS ontology of software testing, as shown in Figure 6, *Capability* includes basic concepts *activity, context, environment*, *method* and *artefacts*. To enable the search of test services using SWS technology, all these aspects of capability must be represented in the structure of Service Profile. We classify the Service Category according to test activity. The test method, context and environment are represented as special input parameters of Profile. The artefacts are represented as the Input and Output of the Profile. The mapping between the concept *capability* in ontology and the Service Profile is shown in Figure 7.

To support flexible service search*, the MorePowerful* relations between capabilities is defined such that capa-

bility C_1 is *more powerful* than C_2 if and only if

a) C_1 and C_2 have the same activities.
b) C_1 and C_2 have the same context.
c) Environment of C_1 is the enhancement of the environment of C_2.
d) The method of C_2 is implemented by C_1.
e) The input artefacts of C_2 are included by input artefacts of C_1 and the output artefacts of C_2 are included by output artefacts of C_1.

The description of tasks is similar to the *Capability*. However, it has different meanings and usages. It describes what is required to be done and specifies how it should be done. Moreover, it includes some meta-data of the test objective such as the services description of the services that to be tested and so on. It is used in the search for testing service and the invocation of testing service. A relation *Capableof* between capability *C* and task *T* is also defined such that *C CapableOf T* means service of capability *C* is capable of performing task *T*.

4.2 Matching of Services

The OWL-S/UDDI Matchmaker [19] is the services capability matching engine. It extends the UDDI Registry and enables the capability search [20] at three levels of matching between capability and request [21].

- *Exact matching*: the capabilities in the registry and in the request match exactly.
- *Plug-in matching*: the service provided is more general than that in the request.
- *Relaxed matching*: there is a similarity between services provided and that in the request.

The Matchmaker provides five filters for users to construct discovery profile: which are *namespace filter*, *domain filter*, *text filter*, *I/O type filter* and *constraint filter* [21]. With these filters, users can construct necessary compound filters to control the precision of matching. The representation of tasks and capabilities as Profiles enables the *Capable Of* relation between capability and task to be implemented by the matchmaker.

The matching engine Matchmaker implemented the capability matching in a general way, the result of the matching may include multiple candidates. Selection from the candidates is based on two considerations. First, the matchmaker tags a score for each candidate service in the result list, the higher the score, the more similar between the candidate and the request. Second, multiple candidates may have the same score, further comparison of the capabilities of the candidates is necessary. The relation *MorePowerful* between capabilities is used here.

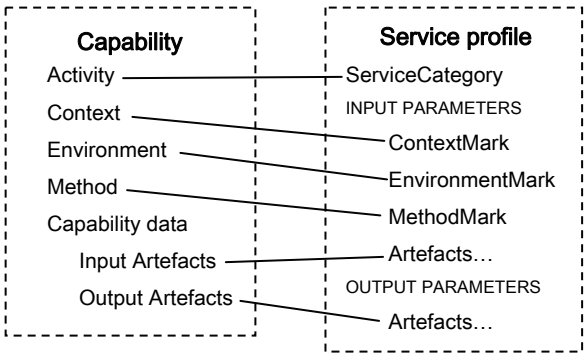

Figure 7. Mapping between Capability and Service Profile

5. Case Study

We have conducted a case study to demonstrate the feasibility of the approach.

In the case study, we wrapped an automated component testing tool CASCAT [22] into a web service. We described its capability in the form of Service Profile as described in the previous section. The Web Service version of the tool is then hosted on a server and opened to the public for invocation. Experiments with search for the service and invocation of the tool as Web Service were carried out successfully.

CASCAT is an automated tool for testing Enterprise Java Beans based on algebraic specification. It can automatically generate test cases from formal specifications written in an algebraic specification language CASOCC. It can also automatically execute the test cases and to check if the equations in the algebraic specification are violated. In the case study, we wrapped the test case generation part to demonstrate the feasibility of the approach. The result Web Service is called *CS* in the sequel.

5.1. Registration

We have built a UDDI registry server using OWL-S/UDDI Matchmaker (Matchmaker). The environment consists of Windows XP running on Intel Core Duo CPU 2.16GHz with Jdk 1.5, Tomcat 5.5 and Mysql 5.0.

The WS *CS* is registered on this UDDI registry. In its Service Profile, the ServiceCategory is "TestCaseGenerationServices". The Input artefact is specified by the class *CasoccSpecification*, which is a subclass of *Specification* and stands for algebraic specification in CASOCC. The context of *CS* is "ComponentTest". Its environment is 'not limited'. Its method is *CASOCC-method,* which is a subclass of *SpecificationBased* method. The output artefact is test case.

The registration of CS is through the Matchmaker Client API with the above as input datum.

5.2 Submitting test tasks

In the experiment, we also built a service that plays the role of test requester. It constructs test tasks and submits them to the test broker which generates requests according to the test tasks and submits them to Matchmaker to search for test services. The particular test task that it produced is to generate test case from CASOCC specification in the context of the test as component test.

5.3. Search and discovery

Once the test broker receives the test task, it generates a capability description from the test task and constructs a Service Profile according to the mapping in Figure 7. It then calls the *API* of the Matchmaker Client to search for test service providers.

5.4. Invocation

To test the invocation of the service, we deployed a Enterprise Java Bean on Jboss platform and wrote a formal specification of the bean in CASOCC. The CS is invoked to generate test case of the component. The result is an instance of the OWL class *TestCase*. The *Location* attribute of the instance contains the URL of the file that contains the test cases generated by the service.

6. Conclusion

In this paper, we presented a service oriented framework for testing Web Services. In this framework, various parties interoperate with each other to complete testing tasks. We adapted ontology to describe the concepts and their relations in the domain of software testing. Based on the ontology, the interoperation between services are specified and implemented in Semantic Web Services technology. The analysis of the framework in a typical scenario shows that the approach has the advantages of supporting dynamic discovery and invocation of testing services without compromise security, privacy and intellectual property rights. It also helps to reduce the risk of unnecessary disturbances to the normal operations of services due to testing activities. The framework is flexible and extendable. Our case study with the WS wrapping of the testing tool CASCAT [22] demonstrated that the framework is technically feasible.

References

[1] Gottschalk, K., Graham, S., Kreger, H. and Snell, J., Introduction to Web Services architecture, At URL: http://www.research.ibm.com/journal/sj/412/gottschalk.html.

[2] Stal, M., Web Services: beyond component-based computing, C. ACM, Vol.45, No.10, 2002, pp71-76.

[3] Berners-Lee, T., Hendler, J. and Lassila, O., The Semantic Web, Scientific American, 2001, 284(5): 34～43.

[4] Zhu, H., A Framework for Service-Oriented Testing of Web Services, Proc. of COMPSAC'06, Sept. 2006, pp679-691.

[5] Chan, W. K., Cheung, S. C. and Leung, K., Towards a metamorphic testing methodology for service oriented software applications, Proc. of QSIC2005, 2005, pp. 470-476.

[6] Tsai, W.T, Paul, R, Wang, Y., Fan, C., Wang, D., Extending WSDL to Facilitate Web Services Testing. Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering, 2002, pp. 171-172.

[7] Looker, N., Xu, J., Assessing the Dependability of SOAP RPC Based Web Services by Fault Injection, Proc. of WORDS'03, IEEE CS Press, 2003, pp163-170.

[8] Jeff Offutt, J., and Xu, W., Generating test cases for Web services using data perturbation, ACM SIGSOFT Software Engineering Notes, Vol. 29, No. 5, 2004, pp127-130.

[9] Jian Bing Li and Miller, J., Testing the semantics of W3C XML schema, Proc. of COMPSAC'05, 2005, pp443- 448.

[10] Uschold, M. and Gruninger, M., Ontologies: Principles, Methods, and Applications, Knowledge Engineering Review, Vol. 11, No.2, 1996, pp93-155.

[11] Zhu, H., Huo, Q. and Greenwood, S., A Multi-Agent Software Environment for Testing Web-based Applications, Proc. of COMPSAC'03, 2003, pp210-215.

[12] OWL Web Ontology Language Reference. at URL: http://www.w3.org/TR/2004/REC-owl-ref-20040210/, 2004.

[13] OWL Web Ontology Language Overview. Available online at URL: http://www.w3.org/TR/owl-features/, 2004.

[14] Zhu, H., Hall, P. A. V. and May, J. H. R., Software unit test coverage and adequacy, ACM Computing Surveys, Vol. 29, No. 4, Dec. 1997, pp336-427.

[15] protégé-owl api, Available online at URL: http://protege.stanford.edu/plugins/owl/api/

[16] Sycara, K., Paolucci, M., Ankolekar, A. and Srinivasan, N., Automated Discovery, Interaction and Composition of Semantic Web services, Journal of Web Semantics, Vol. 1, No. 1, Dec. 2003, pp27-46.

[17] Martin, D., Burstein, M., McDermott, D., McIlraith, S., Paolucci, M., Sycara, K., McGuinness, D., Sirin, E. and Srinivasan, N., Bringing Semantics to Web Services: The OWL-S Approach, at URL: http://www.cs.cmu.edu/~softagents//papers/OWL-S-SWSWPC2004-final.pdf

[18] OWL-S: Semantic Markup for Web Services. at URL: http://www.w3.org/Submission/OWL-S/, 2004.

[19] Paolucci, M., Kawamura, T., Payne, T.R. and Sycara, K., Semantic Matching of Web Services Capabilities, Proc. of ISWC'02, 2002, pp333-347.

[20] Paolucci, M., Kawamura, T., Payne, T.R. & Katia Sycara. Importing the Semantic Web in UDDI, Proc. of Web Services, E-business & Sem. Web Workshop, 2002, pp225-236

[21] Kawamura, T., De Blasio, J-A., Hasegawa, T., Paolucci, M. and Sycara, K., A Preliminary Report of a Public Experiment of a Semantic Service Matchmaker combined with a UDDI Business Registry, Proc. of ICSOC'03, Trento, Italy, Dec. 2003, pp208-224.

[22] Yu, B., Kong, L., Zhang, Y. and Zhu, H., Testing Java Components Based on Algebraic Specifications, Proc. of ICST'08, April 2008, pp190-199.

[23] Srinivasan, N., Paolucci, M. and Sycara, K., An Efficient Algorithm for OWL-S based Semantic Search in UDDI, Proc. of SWSWPC'04, 2005, pp96-110.

[24] OWL-S API, at URL: http://www.mindswap.org/2004/owl-s/api/