Criteria Analysis and Validation of the Reliability of Web Services-oriented Systems

Jia Zhang[1]
Department of Computer Science
Northern Illinois University
jiazhang@cs.niu.edu

Liang-Jie Zhang
IBM T.J. Watson Research
zhanglj@us.ibm.com

Abstract

As Web services become more prevalent, the need to ensure their quality increases. This paper explores the criteria of reliability of Web services-oriented systems, and discusses how to design and generate test cases to conduct tests over Web services. A prototype system is constructed to test the effectiveness and efficiency of our algorithms. The preliminary results show that our approach facilitates the testing of services-oriented systems.

1. Introduction

The Web services paradigm is widely considered as the strategic model for the next generation of distributed computing. However, its extensive acceptance is currently hindered by our lack of technologies to verify the quality of Web services-related software [1]. Although last fifty years of software development history has witnessed the establishment of an independent research branch as *software testing* to guide the quality verification process of a software product [2], we cannot merely apply the traditional software testing technologies to measure and test Web services [3].

The model of Web services poses critical challenges on software testing [4]. Here we just name a few. The fundamental hypothesis of the existing testing methods is that theoretically, it is possible to conduct exhaustive test cases upon the tested software product. It is neither feasible nor practical to apply this assumption to Web services testing. Conducting a large amount of testing on Web services over the Internet may consume significant bandwidth thus be very expensive. In addition, the model of Web services implies that the unique feature of *dynamic discovery and invocation* requires highly efficient testing and selection of Web services components [5-8]. Furthermore, how to design test cases for a Web service using its limited information exposed remains a challenge. Meanwhile, how to mitigate the overhead caused by the Web services-specific transport protocols (e.g., SOAP) deserves further investigation [9].

Due to the specific properties of Web services, these existing software testing models and methodologies deserve re-inspection in the domain of Web services; and new techniques may be required to perform effective and efficient testing and measurements [3]. Software quality generally contains a set of attributes, such as reliability, scalability, efficiency, security, reusability, adaptability, interoperability, maintainability, availability, portability, etc [10]. As the first step, our research targets on how to test the reliability of a Web services-oriented system. By "Web services-oriented system", we refer to a software system consisting of components that will be fulfilled by Web services. Here we adopt the standard definition of *software reliability*: Musa defines *software reliability* as the probability of failure-free operation of a computer program for a specified time in a specified environment [11].

This paper focuses on test cases criteria and generation. The remainder of this paper is organized as follows. In Section 2 we propose our approach. In Section 3 we present the design of our approach. In Section 4 we present the implementation of our approach. In section 5 we present the experiments. In Section 6 we discuss the evaluation of our work. In Section 7 we discuss related work. In Section 8 we draw conclusions and describe future work.

2. Generate Test Cases for Web Services Reliability

Since Web services are remote Web applications hosted by the corresponding service providers, testing has to be conducted remotely by Simple Object Access Protocol (SOAP) messages through Web services' interfaces published using Web Service Description Language (WSDL). In order to eliminate network traffic, the challenge we are facing is that: How to effectively and efficiently select test cases? As the first step to explore how to generate test cases, our goal in this paper is to decide how to generate test cases to test the reliability of a remote Web service based upon its exposed interfaces. Test cases generation based upon local operational

[1] Jia Zhang is also a Guest Researcher of National Institute of Standards and Technology (NIST).

Proceedings of the IEEE International Conference on Web Services (ICWS'05)
0-7695-2409-5/05 $20.00 **IEEE**

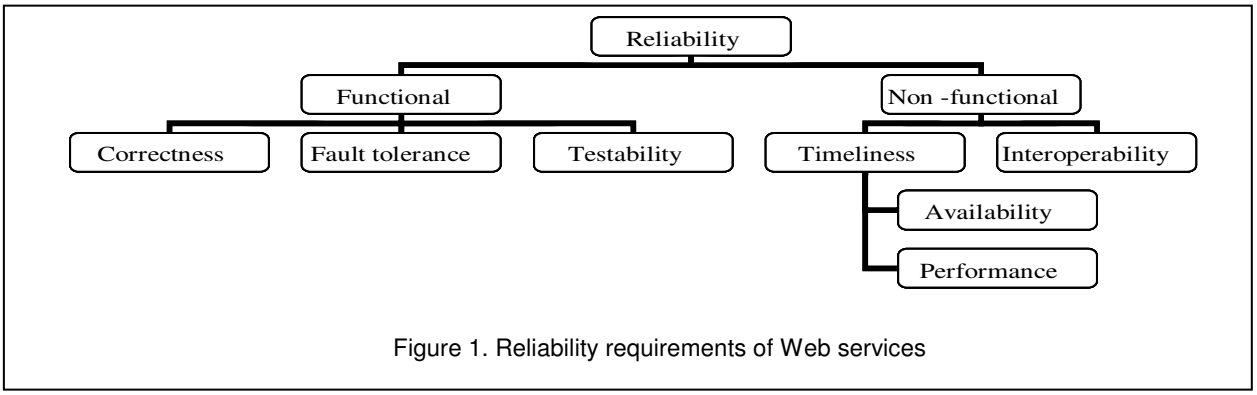

Figure 1. Reliability requirements of Web services

profiles and the final system context are future research topics.

Before we can investigate how to generate test cases, we need to examine which aspects of reliability we are going to test in our context. In other words, what do we mean by reliable Web services?

2.1 Criteria of Web services reliability

The software testing community has provided precise definition for *software reliability*. IEEE defines it as "the probability that software will not cause the failure of a system for a specified time under specified conditions." [12] However, in order to make a Web service meet this definition, more aspects have to be considered due to the unique features of Web services.

We believe that a reliable Web service contains both functional and non-functional requirements, as shown in Figure 1. Along the functional requirements dimension, a reliable Web service must exhibit correctness, fault tolerance, and testability. Correctness here implies that the Web service generates reasonable output from input derived from the problem domain; fault tolerance implies that the Web service is capable of producing acceptable results even though it is faulty; testability implies that the Web service allows existing faults to be detected at testing time. Along the non-functional dimension, a reliable Web service features timeliness and interoperability. Interoperability implies that the Web service coexists with other system components in the context of a specific system environment. Timeliness in turn contains availability and performance. Availability implies that the Web service is available at the invocation time. Performance implies that the Web service delivers at an acceptable speed at the invocation time.

Compared with traditional software reliability [11], reliability of a Web service contains unique requirements of timeliness (availability and performance). Since Web services are remote Web applications that can only be invoked remotely, timeliness is of paramount importance: if a Web service cannot be delivered to a service requester within a predefined time threshold with promised performance, the service will become useless.

Thus, reliability of a Web service can then be defined as a combination of a set of six attributes: correctness (C), fault tolerance (F), testability (T), interoperability (I), availability (A), and performance (P). In other words, the reliability of a Web service *WS* will be a function of the specified six attributes:

$$R(WS) = f(a\mathbf{C}, b\mathbf{F}, c\mathbf{T}, d\mathbf{I}, e\mathbf{A}, f\mathbf{P}) \quad (1)$$

where a, b, c, d, e, and f are quantitative or qualitative measures of particular attributes, which imply that each attribute may contribute differently to the reliability of a Web service in a specific context or scenario. To simplify the problem, in this paper, we only consider correctness and fault tolerance.

2.2. Focus on eliminating Web services candidates

As more and more Web services are published on the Web on the daily basis, it is common that a set of Web services will be found from some public registry with similar functionalities. Instead of proving the correctness of each Web service, our main strategy is to eliminate Web services candidates in terms of their reliability. In detail, if at some point, a candidate Web service fails to meet a predefined lower threshold of reliability requirements, the testing does not need to continue. In other words, our research starts from a list of Web services candidates with similar functionalities. Our goal is to efficiently exclude some candidates using predefined reliability criteria, so that further refined testing can be conducted over the left smaller amount of candidates only.

Carefully designed assertions are used to judge the exclusion process. When assertions cannot be satisfied, the decision can be made safely. It should be noted that the decision is made based upon a predefined threshold instead of the failure of one single test case. We use quantified reliability values as generic assertion standards. Recall from the Formula 1, reliability of a Web service is a function of six attributes. Each test case is designed to

target on one or multiple attributes, and in turn contributes to the result of the reliability function. Note that different attributes may weigh differently for different Web services. Even different test cases targeting the same attribute may weigh differently. Therefore, as each test case is performed, a value will be added[2] to the final value of the reliability function. If at some point, the value over the function exceeds a predefined threshold, which is the assertion to go against, we can stop from further operations. It should be noted that the thresholds are application-specific, as well as the reliability functions.

2.3 Applying boundary values and faulty data to test Web services candidates

Our major strategy is to create test cases to eliminate a Web service candidate. Then the core challenge is that what kind of test cases should be created to efficiently test a Web service candidate. It is obviously neither practical nor feasible to either randomly pick up test cases or try to go through every possible test case from input data domain. Our proposed solution is to utilize boundary values together with faulty data perturbed from boundary values to quickly verify the reliability of a Web service candidate.

Each test case will test a Web service upon a function call which signature contains several parameters. Each parameter requires a specific data type with implicit boundary constraints. Our approach focuses on finding out the boundary values for each input parameter's data type. Let us take a simple example: suppose that a Web service exposes a WSDL interface that includes a string-type parameter defined as follows:

```
<part name="loginId" type="xs:string"/>
```

For this parameter, we can test on boundary values such as: null, "" (empty string), short string (i.e., one character long), very long string (e.g., 200 characters long), string containing "new line" character, non-string value (i.e., integer 3), etc.

For every WSDL interface exposed by a Web service, we will list boundary values for each input parameter. Then we will assemble all boundary values for each input parameter to obtain a list of test cases. For example, suppose a Web service interface contains three input parameters, each one being a string type without further constraints. As discussed above, each parameter can have five boundary values. Assembling them together, we will get a list of fifteen different test cases for the functional call.

[2] This value can be either positive or negative based upon the algorithm used.

In short, our strategy of generating test cases to test the correctness of a Web service is to find boundary values for each parameter. These boundary values are definitely within the input domain. In order to test the fault tolerance of a Web service, we will perturb the boundary values to generate faulty data as test cases. Injecting faulty data to verify fault tolerance is not new. Traditional software testing establishes the fault injection technique [13, 14]. However, applying the traditional fault injection technique to the domain of Web services testing remains challenging. In more detail, we adopted the basic concept of the fault injection; but we need to solve corresponding technical issues.

Here we will first briefly review the concept of fault injection and then discuss the technical challenges. Fault injection is a set of techniques that provide worst-case predictions for how badly a system will behave in the future [13, 14]. It suggests to inject corrupted data to the input of a system, and monitors the output of the system to obtain knowledge of its fault tolerance and interoperability [13]. However, although the basic concept of fault injection technique seems appropriate to be applied to test Web services, how to apply the technique remains a challenge.

The core challenge of the fault injection technique is how to create corrupted data for a testing component. Voas and colleagues proposed to perturb the input domain to find corrupted data [13]. In traditional component-based testing, a testing component is already deployed in its execution environment; thus, it is feasible to conduct arbitrary amount of testing over the testing component. When we deal with Web services, on the other hand, we are facing remote Web components so that network traffic needs to be considered imperatively, let alone the fact that some Web services might have access charges associated. Furthermore, unlike traditional software components, Web services found from public registries oftentimes reveal limited information except the access prototypes defined in WSDL. Therefore, our strategy of designing faulty data to test the fault tolerance of a Web service focuses on perturbing the boundary values for each input parameter's data type. Let us take a simple example: suppose that a Web service login function requires a string-type input parameter with a length limitation of 6 to 16 characters. 6 and 16 character-long strings are both boundary values for the input parameter. Perturbing these two boundary values, we can obtain 5 and 17 character-long strings, which can be used as faulty data to test the fault tolerance of the Web service.

3. Design of generating test cases

In this section we will discuss the detailed design issues and solutions to test case generation. To be specific,

a test case of a Web service is a set of mappings between all published input variables and their values. Each test case can be used to generate a SOAP input message to test the corresponding Web service.

Here we will first briefly introduce the related WSDL specifications on Web services interface definition; then we will discuss how we extract test cases (including both testing data and faulty data) from WSDL definitions.

WSDL is "an eXtensible Markup Language (XML) format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information" [15]. Using the WSDL, a Web service is defined as a set of ports, each publishing a collection of port types that bind to network addresses using common binding mechanism. Every port type is a published operation that is accessible through messages. Messages are in turn categorized into input messages containing incoming data arguments and output messages containing results. Each message consists of data elements; and every data element must belong to a data type, either a XML Schema Definition (XSD) simple type or a XSD complex type[3].

Our basic strategy is to design test cases based upon the boundary values of each formal argument of the published WSDL definition of the Web service. Therefore, our challenge here turns into how to find efficient boundary values for each possible formal argument. Since each input parameter must be a XML-allowed data type, it can be either XML built-in primitive types or user-defined compound types. Let us examine the XML built-in primitive types first.

W3C Specification of the XML Schema language defines nineteen built-in primitive types: string, decimal, boolean, duration, dataTime, time, date, gYearMonth, gYear, gMonthDay, gDay, gMonth, base64Binary, hexBinary, float, double, anyURI, QName, and NOTATION [16]. For each XML primitive type, we extract boundary values based upon: (1) XML constraining facets [16], (2) operational profiles, and (3) semantic meanings.

W3C XML specification defines the constraining facets for each primitive type. Taking *string* as an example, it has the following constraining facets: length, minLength, maxLength, pattern, enumeration, and whiteSpace. We take these defined constraining facets as guidelines to create boundary values. For example, for a WSDL input argument that is an XML data type *string*, we will generate test cases on its length, minimum length, maximum length, null, and empty string, respectively.

XML constraining facets provide generic guidelines for us to find boundary values; and the operational profiles of a Web service will help us find more efficient boundary values. For example, let us consider a login id field with type *string*. From the XML constraining facets of *string*, we know that we need to test on the length of the string. A specific operational profile can help us decide to test whether the string can accept more than 16 characters. In addition, operational profiles can help decide the boundary values for patterns testing, as defined by the corresponding XML constraining facets.

Finally, the semantic meanings of an argument can facilitate boundary values elicitation. Taking an input field of credit card expiration year as an example, it is intuitive for us to test the following cases: whether the input year is a future year or a past year, whether the year is way too far in the future, whether the combination of the year and the month represents a date in the future, whether the month is between 1 to 12, etc.

It should be noted that the first way of boundary value elicitation approach using XML constraining facets can be automated, while the second and third approaches using operational profiles and semantic meanings require manual involvement or more intelligence.

For user-defined compound data types, we can navigate through the hierarchy tree of the data type and design test cases based upon each leaf element that is a XSD simple data type. Figure 2 shows a simplified XSD compound data type AddressBook. The address book for a person contains three elements: her id as a double type, name as a string type, and addresses as a list of address information. Each address is composed of five elements: an *address type* as a double type, an *address line* as a string type, a *city* as a string type, a *state* as a string type, and a *zip code* as a string type. Therefore, there are seven leaf elements in this AddressBook data type: *id, name, addrType, addrLine, city, state,* and *zipCode.* Each element belongs to a XSD simple data type, either *double* or *string.* Then for each element, we can apply our strategy of designing boundary values for XSD simple data types, as we discussed above. Since we prefer to locate errors if there is any, each test case only focuses on testing one edge value of one leaf element, without combining several edge values of multiple elements. In other words, we have purposely limited the edge values to single parameter to avoid the explosion that occurs in the number of combinatorial edge values that could be set at each SOAP input message. Accumulating all of these test cases together, we will obtain a set of test cases targeting at testing the overall AddressBook data type.

After generating test cases based upon boundary values extracted from WSDL definitions, we perturb each value in a test case to generate faulty test cases. For example, if the maximum length of a *string* data type is 16, the generated test case will be a string value with 16-character long; the perturbed faulty test case will be a string value with 17-character long.

[3] Here we omit the fact that WSDL allows other data type system in addition to XSD, since XSD is its canonical type system.

```
<element name="AddressBook" type="tns:AddressBookType"/>
<complexType name="AddressBookType">
  <all>
    <element name="id" type="double/>
    <element name="name" type="string"/>
    <element name="addresses">
      <complexType>
        <all>
          <element name="address" type="tns:Address"
  minOccurs="0" maxOccurs="unbounded"/>
        </all>
      </complexType>
    </element>
  </all>
</complexType>
<complexType name="Address">
  <all>
    <element name="addrType" type="double"/>
    <element name="addressLine" type="string"/>
    <element name="city" type="string"/>
    <element name="state" type="string"/>
    <element name="zipCode" type="string"/>
  </all>
</complexType>
```

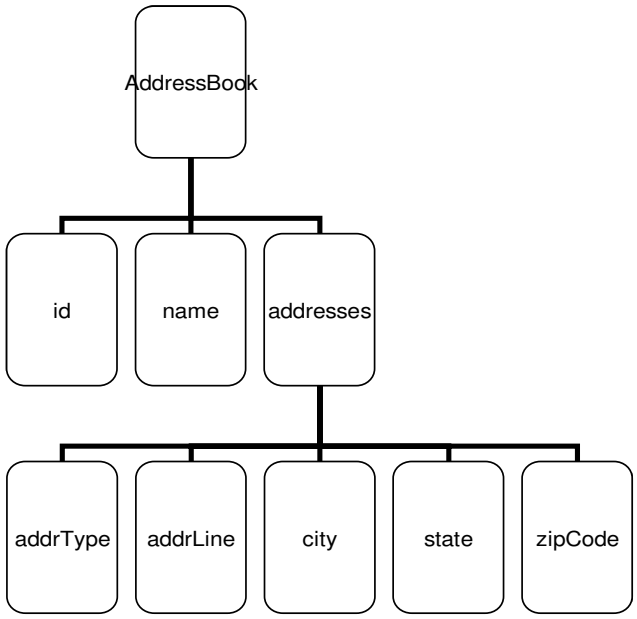

Figure 2. A simple example of XSD complex data type AddressBook

4. Implementation

We implemented a prototype that realizes the reliable Web services selection, as shown in Figure 3. There are six components and four databases. The six components are: a test manager, a test case generator, a testing agent creator, a Web service searcher, a reliability function editor, and a system tester WS-Net. The four databases are: a candidate database, a test case database, a system test case database, and a log database. In Figure 3, dotted lines represent control information, and solid lines represent data interactions. Numbers associated with lines represent the order of the operations that are performed by the system in a process of selection. We will walk through a complete procedure and discuss included components.

The test manager is the central coordinator of the system. It directly controls the other three components: the test case generator, the test agent creator, and the system tester. The test manager will first start the test case generator, which is responsible of creating all test cases. The test case generator identifies test cases. The locations and the binding information of the candidates are stored in the candidates database. The generated test cases are also stored in the test case database. In our prototype, our test case generator automates test case generation using the XML constraining facets approach discussed in Section 3.

After the initial set of test cases are generated and stored in the corresponding databases, the test manager initiates the test agent creator. Each Web service candidate will be assigned a dedicated test agent. For each Web service candidate, the test agent creator creates a test agent, equipped with test data, assertions, and knowledge from the test case database. A proxy is also created for each test agent to receive information from the agent and monitor its execution. Then the test agent can start to test the remote Web service candidate site, using the location and binding information from the candidate database. The testing results are stored in the log database. By assigning dedicated test agent to each Web service candidate, testing over multiple candidates can be conducted parallel.

After a testing agent starts its testing work, the test manager starts the system tester component, which is in charge of system testing. The system tester then creates a substitute for every component in the system by using information from the log database. Finally the system tester initiates the WS-Net [17] to run the tests, loading test cases from the system test case database. WS-Net is yielded from our previous research, which is a Petri nets-based tool to simulate and validate the cooperation of multiple Web services.

5. Experiments

In order to examine the effectiveness of our proposed test agents-based testing and selection approach, we carried out a series of simulations to study the system performance. An environment was set up to simulate a student registration and record system, where students can register courses and retrieve course grades on line. The system was designed to be composed of a sequence of components, each being implemented by a Web service.

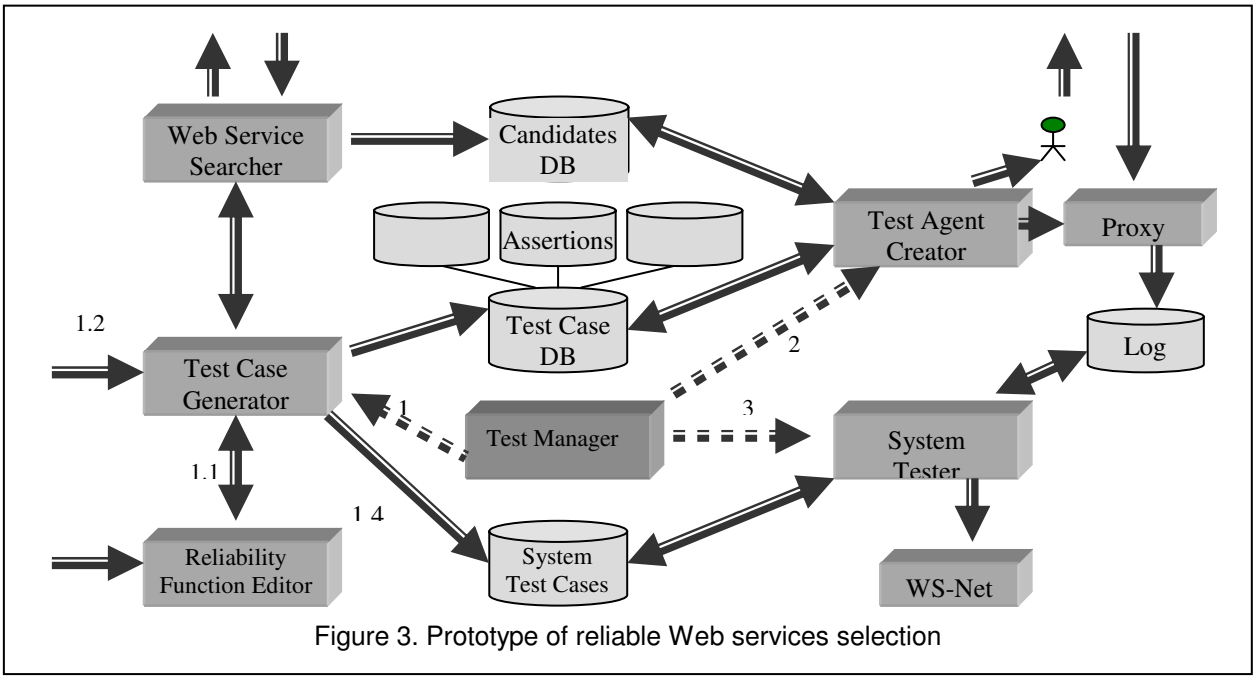

Figure 3. Prototype of reliable Web services selection

The requirements of each component were pre-defined; and the interaction process of the whole system was pre-defined. A component X intends to receive a set of input arguments – a student id (string), course id (string), term (string), and year (integer) – and checks databases to return the corresponding course credit hours (integer) and the grade (double). Its detailed WSDL definition was presented and discussed in Section 4. Three Web services candidates Y_1, Y_2, and Y_3 implemented the same WSDL definition.

To build three Web services candidates, we (1) obtained three copies of the module, (2) injected errors to the first two copies, (3) deployed the three copies to three machines, and (4) wrapped each one with a Web service interface. The errors we injected include: to make the module crash upon receiving an empty string as the student id, to remove semantic checking of the input term and year, etc. The third candidate is error-free, so that it is the one that should be selected.

We tested the efficiency of our boundary value-based test cases and faulty test case generation algorithm. In Y_1 and Y_2, we embedded various types of errors, such as computational faults, XML and SOAP processing faults, data exception handling faults, etc. We perform three categories of test case generation: (1) manually and randomly pick up test cases from the input domain, (2) manually go through possible test case from input data domain, and (3) automatically generate test cases using our boundary value-based approach. The number of test cases in the second category is N times of the maximum of the number of test cases in the first and third categories.

We tested many groups of experiments by embedding various numbers of errors. We found that random test case selection is the least robust algorithm to find errors. Based upon the chosen number of number N, when almost all of the possible test case is covered, the second exhaustive approach can find all errors. However, it should be noted that both algorithms have to go through manual process.

We found that our boundary value-based test case generation approach is efficient to find computational faults, XML and SOAP processing faults, as well as data exception handling faults.

In order to further test the efficiency of three algorithms, we chose to set up the reliability decision threshold to zero, which means that all test cases will be conducted. The testing results are similar to that was described above.

In summary, our experiments show that our test case generation is efficient.

6. Evaluations

By constructing test data including normal data and corrupted data, our approach is capable of certifying whether the tested Web service thoroughly fulfill the functional requirements as desired. In addition, by perturbing the test data to imitate unusual events, our approach is capable of testing whether the hosts of Web services act maliciously or errantly at invocation times.

Based upon remote Web services' published descriptions and service requester's operational profiles, currently our test agents can test the functionalities of the Web services and their reliability. However, our method has the potential to test more non-functional attributes of Web services. For example, test agents can conduct a matrix of test data to a remote Web service to test more

attributes.

By perturbing the output data of a Web service gathered by test agents in a system integration test, our approach is capable of testing whether a Web service's errant action will affect the quality of a composed environment.

In addition, it should be noted that, although our method concentrates on testing the reliability of Web services as components in an application system, this method can be used to test the reliability of standalone Web services. That being the case, the second level of interoperability testing can be omitted.

Furthermore, although our current research focuses on how to assist service requesters to select reliability-aware Web services components, this research has the potential to be applied to facilitate Web services certification processes.

In summary, our approach provides a cost-efficient method to reveal the testability of remote Web services components. Our approach also provides an approach to facilitate service requesters to make better decisions.

7. Related Work

Casati et al. suggested that Web service providers define *service quality metrics*, which contain non-functional parameters specifying the cost, duration, and other characteristics of a service, to help service requestors make decisions over multiple candidates [18]. However, their work remains as a high-level abstraction without technical discussions such as how to construct *service quality metrics*.

Maximilien and Singh proposed to adopt dedicated agents to gather, store, aggregate, and share QoS data in order to help dynamic service selection [7]. Contrast with their work, we use agents for different purposes. Their agents serve as independent information centers that provide QoS data for service requestors; our agents are dynamically created by service requestors and travel to service candidates to gather QoS information at run time.

Menascé presented a way to calculate response time for composite Web services on the perspective of service requestors [19, 20]. Since our research concentrates on efficient testing methods of Web services, we only consider response time for a single Web service component.

Zhang et al. built an XML-based search engine for appropriate Web services from public entry [21]. Our work starts from a list of Web services candidates pre-selected by such a kind of method.

Offutt and Xu proposed to adopt data perturbation technique to generate test cases of testing message communications between pairs of Web services. In contrast with their approach, our research aims to help service requestors automatically create test cases to select Web services found from public registry.

OASIS' WS-Reliability [22] provides a standard to ensure reliable message interactions between Web services. Contrast with this standard that targets on reliable messaging at the transport level, our research targets on analyzing and testing reliability of a Web service itself.

8. Conclusions and Future Work

This paper explores the criteria of reliability of Web services-oriented systems, and discusses how to design and generate test cases to conduct tests over remote Web services.

Our future work will include exploring more selection criteria to ensure more non-functional attributes and conducting more case studies.

9. References

[1] N. Leavitt, "Are Web Services Finally Ready to Deliver?" *IEEE Computer*, 2004: pp. 14-18.

[2] M.A. Friedman and J.M. Voas, *Software Assessment: Reliability, Safety, Testability*, 1995, New York: John Wiley & Sons, Inc.

[3] J. Zhang and L.-J. Zhang, "Web Services Quality Testing", *International Journal of Web Services Research*, 2005. 2(2): pp. 1-4.

[4] T. Tsai, R. Paul, Z. Cao, L. Yu, A. Saimi, and B. Xiao, "Verification of Web Services Using an Enhanced UDDI Server", *Proceedings of the Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, Guadalajara, Mexico, Jan. 15-17, 2003, pp. 131-138.

[5] D.A. Menascé, "Composing Web Services: A QoS View", *IEEE Internet Computing*, 2004: pp. 88-90.

[6] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q.Z. Sheng, "Quality Driven Web Services Composition", *Proceedings of the twelfth ACM International Conference on World Wide Web*, Budapest, Hungary, 2003, pp. 411-421.

[7] E.M. Maximilien and M.P. Singh, "A Framework and Ontology for Dynamic Web Services Selection", *IEEE Internet Computing*, 2004. 8(5): pp. 84-93.

[8] W.T. Tsai, Y. Chen, R. Paul, N. Liao, and H. Huang, "Cooperative and Group Testing in Verification of Dynamic Composite Web Services", *Proceedings of 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - (COMPSAC)*, Sep., 2004, pp. 170-173.

[9] C. Werner, C. Buschmann, and S. Fischer, "WSDL-Driven SOAP Compression", *International Journal of Web Services Research*, 2005. 2(1): pp. 14-35.

[10] P. Neumann, "Principled Assuredly Trustworthy Composable Architectures, emerging draft of the final report for DARPA's Composable High-Assurance Trustworthy Systems (CHATS) program, 2004", 2004, http://www.csl.sri.com/users/neumann/chats4.pdf.
[11] J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability Measurement Prediction Application*, 1987: McGraw-Hill.
[12] IEEE, *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*. 1988, IEEE Std 982.1-1988, Institute of Electrical and Electronics Engineers.
[13] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*, 1998: New York: John Wiley & Sons, ISBN 0-471-18381-4.
[14] J. Voas, "Certifying Off-The-Shelf Software Components", *IEEE Software*, 1998: pp. 53-57.
[15] WSDL, 2004, http://www.w3.org/TR/wsdl.
[16] W.C.D. Types, http://www.w3.org/TR/xmlschema-2/.
[17] J. Zhang, C.K. Chang, J.-Y. Chung, and S.W. Kim, "WS-Net: A Petri-net Based Specification Model for Web Services", *Proceedings of IEEE International Conference on Web Services (ICWS), IEEE CS Press*, San Diego, CA, USA, Jul. 6-9, 2004, pp. 420-427.
[18] F. Casati, M. Castellanos, U. Dayal, and M.-C. Shan, "Probabilistic, Context-sensitive, and Goal-oriented Service Selection", *Proceedings of the 2nd ACM International Conference on Service Oriented Computing*, New York, NY, USA, 2004, pp. 316-321.
[19] D.A. Menascé, "QoS Issues in Web Services", *IEEE Internet Computing*, 2002. 6(6): pp. 72-75.
[20] D.A. Menascé, "Response Time Analysis of Composite Web Services", *IEEE Internet Computing*, 2004. 8(1): pp. 90-92.
[21] L.-J. Zhang, T. Chao, H. Chang, and J.-Y. Chung, "XML-based Advanced UDDI Search Mechanism for B2B Integration", *Electronic Commerce Research Journal*, 2003. 1(3): pp. 25-42.
[22] WS-Reliability, 2004, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm.