An Approach to Facilitate Reliability Testing of Web Services Components

Jia Zhang[1]
Department of Computer Science
Northern Illinois University
Chicago, IL 60115
jiazhang@cs.niu.edu

Abstract

The paradigm of Web services that transforms the Internet from a repository of data into a repository of services has been gathering significant momentum in both academia and industry in recent years. However, as more and more Web services are published on the Internet, how to choose the most appropriate Web service components from the sea of ever-changing, unpredictable, and largely uncontrollable Web services poses a big challenge. In this paper we propose a mobile agent-based approach that selects reliable Web service components in a cost-effective manner.

1. Introduction

The paradigm of Web services has been gathering significant momentum in both academia and industry in recent years. This paradigm transforms the Internet from a repository of data into a repository of services. Simply put, a Web service is a programmable Web application that is universally accessible through standard Internet protocols [1]. By means of each organization exposing its software services on the Internet and making them accessible via standard programmatic interfaces, this model of Web services offers a promising way to facilitate Business-to-Business (B2B) collaboration. In addition, Web services technology provides a uniform framework to increase cross-language and cross-platform interoperability for distributed computing and resource sharing over the Internet. Furthermore, this paradigm of Web services opens a new cost-effective way of engineering software to quickly develop and deploy Web applications by dynamically integrating other independently published Web services as components to conduct new business transactions. It is predicted that the Web service market would grow to $28 billion in sales in these several years [2].

However, it is not clear that this new model of Web services provides any measurable increase in software trustworthiness, which is coined to represent people's confidence in software products [3] and can be assessed by the set of classic software "ilities," such as reliability, scalability, efficiency, security, usability, adaptability, maintainability, availability, portability, etc [4]. The essential feature of "dynamic discovery and integration" of Web services model, among other aspects, poses new challenges to software trustworthiness. In a traditional software system, all of its components and their relationships are pre-decided before the software runs. Therefore, each component can be thoroughly tested, and the interactions among the components can be fully examined, before the system starts to execute. Web services extend this paradigm by providing a more flexible approach to dynamically locate and assemble distributed Web services in an Internet-scale setting. More precisely, when a system requires a Web service component, the system will search a public registry where Web services providers publish their services, choose the optimal Web service that fulfills the requirements, bind to the service's Web site, and invoke the Web service. In other words, in this dynamic invocation model, it is likely that users may not even know which Web services will be used until run time [5], much less those Web services' trustworthiness (i.e., their "ilities"). Consequently, how to select qualified Web services remains a challenge. Even worse, since Web services are hosted by their own providers over the Internet, remotely testing the trustworthiness of Web services via constant binding is neither efficient nor effective. Furthermore, how to test the interoperability of a remote Web service in a specific software environment remains another challenge.

In summary, the flexibility of Web services-oriented computing is not without penalty since the value added by this new paradigm can be largely defeated if: (1) the selected Web service components do not thoroughly fulfill the requirements (i.e., functionally or non-functionally), (2) the hosts of Web services components act maliciously or errantly at invocation times, (3) erratic Internet behaviors or resource scarcity pose unendurable time delays, or (4) the selected Web services components act errantly in the composed environment.

Meanwhile, Web services technology is still in its infancy, and the trustworthiness has not gained significant attention. To date, current research is still preoccupied by

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

Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)
1071-9458/04 $ 20.00 IEEE

low level technical mechanisms of Web services, e.g., how to publish a Web service, how to compose Web services, what is the overall architecture of Web services-oriented system, etc. As Web services become more mainstream, Web services trustworthiness will hinder the adoption of Web services. The need to ensure the trustworthiness in loosely coupled Web services that need to be integrated in a seamless fashion requires methodologies beyond the current state-of-the-art in this field. Therefore, it is promising if researchers start to explore how to measure, test, and enhance the trustworthiness of Web services-oriented systems, so that related techniques can be investigated in the right context.

In practice, a fundamental question needs to be answered first: whether the trustworthiness of Web services is really measurable and testable, given the fact that it is difficult to quantify people's confidence in software product? Our position is that, since general software trustworthiness can be viewed as containing a number of ilities [4], if each ility of a Web service-oriented system is measurable and testable, then the trustworthiness of the whole system is measurable and testable. In other words, if a Web services-oriented system scores high in every ility[2], it is safe to say that the trustworthiness of the system is high.

Therefore, we believe that a feasible strategy is to first investigate each individual ility in the domain of Web services independently before exploring the mixed ilities. As the first step, in this paper, we will focus on the techniques of ensuring the reliability of a Web services-oriented system by selecting appropriate Web services components in a cost-effective manner. By "Web services-oriented system", we mean a software system that consists of components that will be fulfilled by Web services. The term "appropriate" here implies that the Web service: (1) fulfills the functional requirements; (2) does not act errantly (e.g., does not generate some unexpected results); and (3) fits in the system (i.e., its output will not cause undesirable problems if the system runs). We will mainly address the fourth problem (i.e., the selected Web services components act errantly in the composed environment) and partially the first problem (i.e., the selected Web service components do not thoroughly fulfill the requirements, functionally and non-functionally). More specifically, this paper addresses the following three research questions:

Question 1: *How to effectively and efficiently test remote Web services?*

Question 2: *How to test the reliability of remote Web services?*

Question 3: *How to test the interoperability of a remote Web service in a specific software environment?*

To provide a solution to these three questions, this paper proposes a mobile agent-based approach to assist a service requester to select appropriate Web services components. Our approach provides a more cost-efficient method to help service requesters make better decisions. It should be noted that this paper focuses on the generic framework and techniques of our approach, rather than the experiments and the related assessments. The remainder of this paper is organized as follows. In Section 2 we discuss the strategy and technology considerations. In Section 3, we present our approach. In Section 4, we discuss the merits and limitations of our approach. In Section 5, we discuss related work. In Section 6, we draw conclusions and describe future work.

2. Strategy and Technology Considerations

In this section, we will discuss the strategy and technology considerations. Before we start, we will briefly introduce the core techniques and standards of Web services to provide readers with some background context. A Web service is a programmable Web application that is universally accessible through standard Internet protocols [1]. Web services typically adopt a provider/broker/requester architectural model [6]. A service provider registers Web services at service brokers; a service broker publishes registered services; and a service requester searches Web services from service brokers. The essential aspect of this model is the concept of dynamic invocation: Web services are hosted by service providers; and service requesters dynamically invoke the Web services over the Internet upon an on-demand basis.

The paradigm of Web services mainly embraces three core categories of supporting facilities: communication protocols, service descriptions, and service discovery [7]. Each category possesses its own *ad hoc* standard: the Simple Object Access Protocol (SOAP) [8] acts as a simple and lightweight protocol for exchanging structured and typed information among Web services; the Web Service Description Language (WSDL) is an XML-based description language that is used to describe the programmatic interfaces of Web services [9]; and the Universal Description, Discovery, and Integration (UDDI) standard [10] provides a mechanism to publish, register, and locate Web services. It should be noted that here we adopt a narrow definition of Web services that refers to an implicit definition of SOAP+WSDL+UDDI, due to the fact that these three standards are the extensively accepted industrial standards up to date.

[2] Here we omit the exception that some ilities naturally conflict with each other, such as the fault tolerance and testability.

With the basic background, we can now define the problem domain more specifically by depicting a typical problem scenario, so as to facilitate later discussions. Suppose in a project there are four serially connected components, say X_1, X_2, X_3, and X_4, as shown in Figure 1. To simplify the question, circular dependency relationship is not considered, i.e., the output of the component X_4 does not directly or indirectly affect the input of the component X_1. We also assume that the requirements of each component have been defined. The subscriptions of the components also represent the dependencies between the components. For example, the output of the component X_1 is the input of the component X_2; the output of the component X_2 is the input of the component X_3; and so on. Note that to simplify the question, we only consider a serial dependency relationship between components here without considering a parallel relationship. Assume that it has been decided that the component X_2 will be implemented by Web service X_2; and the current task is to search for a suitable Web service to fulfill the requirements of the component X_3. After searching a UDDI [10] public registry using the functional requirements of the component X_3, a set of Web service candidates are found, say, Y_1, Y_2, …, Yn. These candidates are all published using the *ad hoc* industry standard WSDL [9] by different service providers thus may possess different qualities. Now the problem what we are facing is that: how to select an appropriate Web service from this list (i.e., Y_1, Y_2, Y_3, …, Yn) to fit the position of X_3, in order to ensure the reliability of the project?

2.1. Applying mobile agents to test Web services

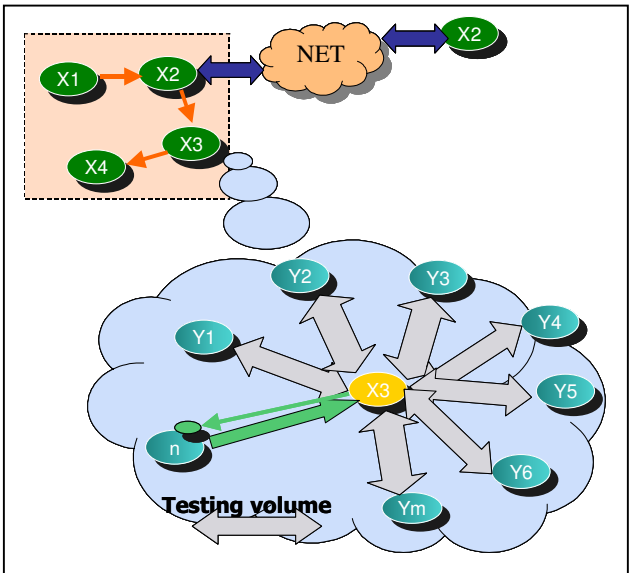

Figure 1. Web services-oriented scenario

In order to select an appropriate Web service, the candidate Web services should be independently tested before a decision is made. Whether the testing stops as soon as a qualified candidate is found or until all candidates are tested is beyond the scope of this paper. What we are interested here is how to test each candidate. Since Web services are remote Web applications hosted by the corresponding service providers, testing has to be conducted remotely by SOAP messages through Web services' interfaces published using WSDL. Nevertheless, if each test case is performed remotely over the Internet, as shown in Figure 1, the testing volume can be significant thus generates enormous network traffic. Moreover, if each test case is conducted remotely, how to ensure the trustworthiness of the corresponding pair of SOAP request and SOAP reply messages (e.g., not to be maliciously attacked in the process of Internet transport) can produce significant overhead.

In order to eliminate both network traffic and transport protection overhead, we explore the mobile agent technology to test Web services. Here we will first briefly introduce the concept of mobile agent paradigm. Mobile agents refer to self-contained and autonomous software programs that can move from one computer to another through the network environment and act on behalf of users or other entities [11-13]. Utilizing the migratory characteristics of mobile agents, we apply mobile agents to conduct Web services testing. More specifically, in order to test a remote Web service, a mobile agent is generated and migrates to the remote Web service candidate site carrying test cases. Then the mobile agent will conduct all the testing on the remote Web service site, and the testing results will be passed back, as shown in Figure 1. If the number of original SOAP request messages using traditional methods is N, when the number of test cases becomes significant, the network traffic can be reduced to N/2 by using the mobile agent, as shown in Figure 1. In addition, from the Web service providers' perspective, they only need to authenticate the mobile agent once when it arrives, instead of authenticating each incoming SOAP request message. Furthermore, since the testing tasks are embedded into the mobile agents that are then dispatched to remote Web service sites and operate asynchronously at the remote sites, the service requesters can be released from being forced to remain the bindings with every candidate Web service.

Meanwhile, in order for a Web service to be adopted as a system component, its system interoperability needs to be fully tested. In other words, it is imperative that the testing Web service component coexists with other system components in the context of the specific system environment. In other words, it is necessary to investigate how the whole system will react with every possible output from the testing Web service component. As we

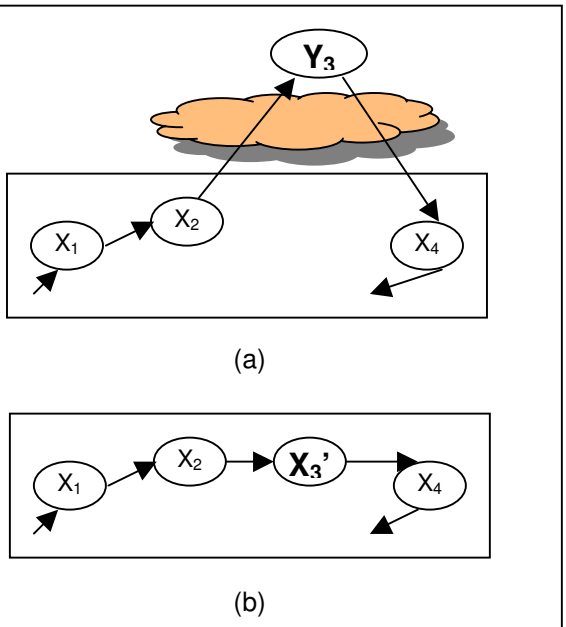

Figure 2. Interoperability testing

discussed, it is neither efficient nor practical for service requesters to bind to the service providers to examine every possible output, as shown in Figure 2(a), using the scenario we discussed earlier where Y_3 is a remote Web service candidate. To avoid generating significant Internet traffic and facilitate the test process, we utilize mobile agents for another purpose, as shown in Figure 2(b). By sending a mobile agent to the remote Web service site Y_3 to perform a set of testing, the test results can also be gathered by the mobile agent. If the set of test data is comprehensive enough in respect to the real operational profiles, which refer to the possible execution scenarios in the context of the final system environment, the collection of the tuples of the test data and the corresponding test results can be used to simulate a substitute component X_3' for the remote Web service Y_3 in the final system, as shown in Figure 2(b). Here we use the term substitute because from the system's perspective, the collection of the tuples produces exactly the same results upon the same input data, thus acts exactly the same as the real remote Web service. In other words, we intend to employ mobile agents to carry back the full states of the remote Web service candidate so as to perform system interoperability testing locally, as shown in Figure 2(b). The phrase *states of a Web service* here refer to the collection of the tuples of the input and the corresponding output data of the Web service over the entire input data space, which can be denoted as follows:

$$S(ws) = \bigcup_{i=1}^{n} (x_i, y_i), \quad y_i = f(x_i), \quad x_i \in X$$

where *S* represents the states of the Web service named *ws; f* represents the functionality of the Web service, which generates a unique output over an input; and **X** represents the entire input space of the Web service.

Now that we have decided to exploit the mobile agent technique to test remote Web services and facilitate Web service interoperability testing, our next step is to decide how to equip mobile agents to test remote Web service candidates and produce their substitutes. We will explore the fault injection technique to provide a solution and it will be discussed in detail in the next section.

2.2. Applying fault injection to Web services

In the last section, we propose that mobile agents will be created and carry test cases to remote Web service sites to conduct tests there. Then the next question we are facing is that: (0) how to effectively and efficiently prepare the test cases for the mobile agents to carry? Since it is obviously impractical to test every piece of datum in the possible input space, the question can be broken down into two questions: (1) how to decide possible test case space; and (2) how many test cases are sufficient and necessary to obtain the full state of a remote Web service.

In order to answer these two questions, let us reexamine the goal of the test cases that we are interested. As we discussed earlier, here the test cases intend to carry back the full states of the remote Web service. The full states of a Web service are based upon the entire input space, which ought to involve the input data able to test both the functional and non-functional attributes of the Web service. The test data related to the functional attributes intend to test whether the Web service fulfill the functional requirements; and the ones related to non-functional attributes intend to test the ilities of the Web service, which in the context of the specific scenario in our paper imply the reliability of the Web service, and more specifically its interoperability.

Oriented to the functional attributes of a Web service, we design the test data based upon the WSDL definition of the Web service. Here, we will focus on the strategy that we considered to design the test cases targeting the reliability of a Web service. We view the reliability of a Web service from two perspectives: the fault tolerance of the Web service individually and the fault tolerance of the Web service as a component in the context of the system environment (i.e., the environment described in Figure 1).

In order to facilitate the reliability testing, we explore the fault injection technique. We will first briefly review the concept of fault injection. Derived from the technique used in traditional industry for a long time (e.g., automobile manufacture), fault injection is a set of techniques that provide worst-case predictions for how badly a system will behave in the future [14, 15]. More specifically, the Interface Propagation Analysis (IPA)

technique proposed by Voas and colleagues is an advanced fault injection technique to test upon black-box like software systems [16]. We believe that IPA is a right candidate to test the interoperability of Web services: similar to normally called Commercial-off-the-shelf (COTS) components, users of Web services have no access to their internal source code. Users can only access Web services via Simple Object Access Protocol (SOAP) [8] request messages, and get results from Web services via SOAP response messages. Therefore, Web services exhibit as black-box systems to the users' perspectives.

The IPA technique injects corrupted data to the input of a black-box system [14], and monitors the output of the system to obtain knowledge of its fault tolerance, as shown in Figure 3 (a). Therefore, IPA can help us test the vulnerability of a Web service serving as a component in a software system with respect to two levels: (1) the Web service in isolation, and (2) the Web service as a component interoperating with other parts of the system. The second level should be considered along with two scenarios: (2.1) when the Web service component returns corrupt information or no information at all, and (2.2) when the Web service fails to interoperate with other components of the system. In summary, IPA can be applied to test the degree of how a system can tolerate the Web service.

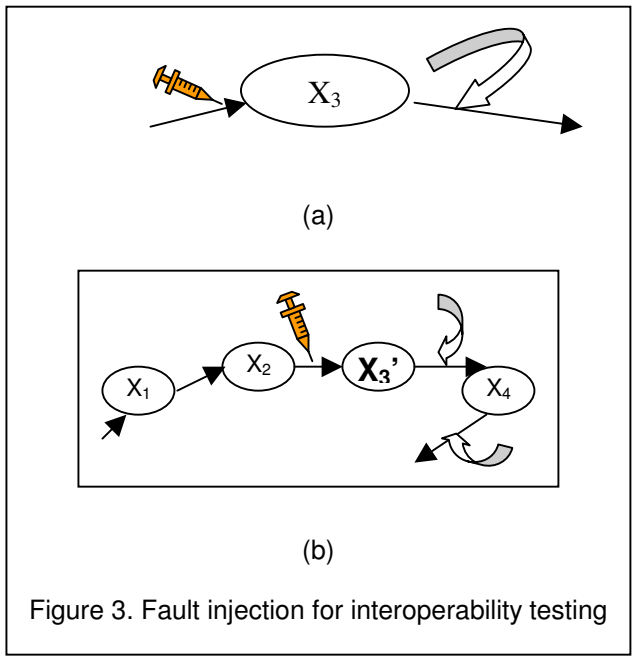

Figure 3. Fault injection for interoperability testing

Although the concept of IPA seems appropriate to test the interoperability of Web services, how to apply the IPA technique in the domain of Web services remains challenging. The core issue of the IPA technique is how to produce corrupted data for the tested component. Voas and colleagues propose to perturb the input domain to find corrupted data [14]. However, in traditional component-based testing, the testing component is already deployed in the execution environment; thus, it is feasible to conduct arbitrary amount of testing over the testing component. As shown in Figure 3(b), when we test the interoperability of a remote Web service, as we discussed earlier, we will test the interoperability of a substitute X_3' of the remote Web service. According to the concept of IPA, faulty data should be injected into the X_3', then we will monitor the output of X_3' and the output of its successor X_4, and so on, as shown in Figure 3(b). However, since X_3' is a substitute, in order to enable the testing, X_3' should already hold the corresponding output value that the remote Web service will produce for the expected faulty data. Furthermore, in order to monitor the output of the successor X_4, the output of X_3' will be the input of X_4. In other words, in order to facilitate the local interoperability testing, the corresponding mobile agent should not only carry the test data from the normal input domain specified by the WSDL interface of the Web service, but also all the expected faulty input data to acquire a comprehensive state of the remote Web service.

The faulty data should be divided into two sets: (1) to test the Web service in isolation, as shown in Figure 3(a); and (2) to test the Web service as a component in the system environment, as shown in Figure 3(b). In order to test the vulnerability of a Web service in isolation, we can simulate the scenario when the data sent to the Web service is corrupted, and monitor and analyze the postcondition of the Web service to decide whether the output events from the Web service is undesirable. To decide whether the Web service is vulnerable with some unexpected events, a global perspective needs to be considered, such as the Web service's functional correctness, the output of the Web service, and the output effects on the system environment. In addition, certain amount of such testing should be performed to achieve particular level of assurance.

Without the ability to automatically simulate the fault injection to the system model, the system integrator would be burdened with the responsibility of manually introducing the corrupted data and recording the propagation results. Our previous research yields a Color Petri Net (CPN)-based specification model for Web services, which is called WS-Net [17]. Running this executable architectural model, corrupted data can be fed into the output port of component X_1 and output port of component X_2', respectively. Examining the output port of the component X_3 and other components can obtain the propagation results.

2.3. Applying assertions to Web services components selection

When mobile agents are dispatched to remote Web

services sites, they will conduct test cases on behalf of the service requester. Since we are only interested in Web services candidates that meet all the requirements, if at some point, a mobile agent at one candidate site finds that the Web service fails to meet some of the requirements, the testing does not need to continue. In other words, if we are in some way capable of making decisions before all the testing is completed that a candidate Web service is not the one we are looking for, the rest of the testing tasks can be discarded. This strategy can potentially largely benefit both the service requester and the service provider. From the service requester's perspective, this method can shorten the decision time and lessen the number of mobile agents the requester needs to monitor. From the service provider's perspective, this strategy can alleviate traffic so that the Web service may support more simultaneous access.

In order to find out when the decision can be made before the whole set of test cases are completed, we choose to use assertion technique. Software assertions are Boolean functions that evaluate to be TRUE when a program state satisfies some semantic condition, and FALSE otherwise [18]. If an assertion evaluates to be FALSE, it will be considered the same as if the execution of the program resulted in failure, even if the output for that execution is correct. Carefully designed assertions are embedded into mobile agents' code to migrate to the tested Web services sites. When assertions cannot be satisfied, the decision can be made safely. Contrast to the fact that assertions have been extensively used to prove program correctness, we choose to apply assertions to increase trust in Web services components selection and facilitate the selection process.

3. Mobile Agents-based Approach

Since we have decided the techniques we want to use, in this section we will present the procedure of our mobile agents-based approach for efficient reliable Web services components selection. Our approach is illustrated in Figure 4. We will walk through the scenario step by step.

Step 1: The service requester creates a mobile agent. A database is also prepared for the mobile agent, which includes two set of information: test data and assertions. The test data contains two sets of data for two purposes. One is the data set to test the functionality of the Web service, which is constructed according to the operational profile of the service requester and the WSDL description of the Web service. The other one is the data set to test the vulnerability of the Web service, which is constructed according to the fault injection technique.

The assertions are embedded into the mobile agent to test the trustworthiness of the Web service. These assertions are designed according to the operational profile of the service requester and the functional requirements of the desired component. In addition, a stub is created at the service requester site for the mobile agent,

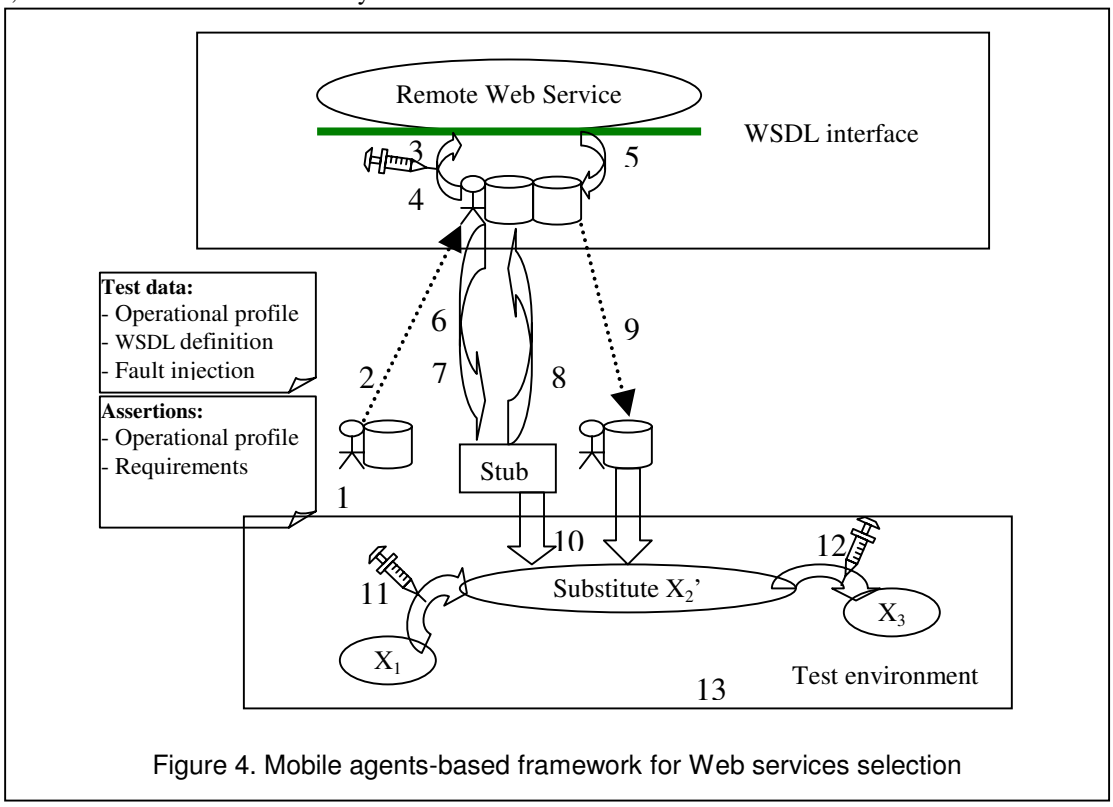

Figure 4. Mobile agents-based framework for Web services selection

which functionality is to monitor the assertions sent back from the mobile agent, and decide how to act accordingly. If the mobile agent does not send information back for a predefined time period, the stub will actively query the mobile agent to ensure that it is still alive.

Step 2: The service requester dispatches the mobile agent to the remote Web service site.

Step 3: The mobile agent generates SOAP request messages to send to the Web service to test the functionality of the Web service, using the test data from the database associated with the mobile agent.

Step 4: The mobile agent generates SOAP request messages to send to the Web service to test the vulnerability of the Web service, using the test data from the database associated with the mobile agent. This step injects some faulty data to the Web service to test how the Web service will act.

Step 5: The mobile agent receives the SOAP response messages sent by the Web service for every test data.

Step 6: The mobile agent tests the assertions for the functionality and vulnerability of the Web service. Assertions results are stored into the log database associated with the mobile agent, and forwarded back to the corresponding stub at the service requester site.

Step 7: The mobile agent tests the assertions for its own trustworthiness at the moment. Assertions results are stored into the log database associated with the mobile agent, and forwarded back to the corresponding stub at the service requester site.

Step 8: The stub analyzes the assertions returned from the mobile agent. If the stub finds that the Web service does not meet the requirements of the desired service (i.e., functional requirements and vulnerability requirements), it will send a command to the mobile agent to terminate its tasks, and remove the Web service from the service requester's candidate list. The process will end afterwards. If the stub finds that the mobile agent has been maliciously attacked, it will send a command to the mobile agent to terminate its tasks, records the information, and goes back to Step 1 to create another mobile agent. If the stub finds that similar situation happens repeatedly (e.g., the number of times that the mobile agent fails for the same reason of malicious attacks surpasses some predefined threshold), it will consider the Web service as untrustworthy and removes it from the service requester's candidate list.

Step 9: Eventually the mobile agent finishes all the test cases at the remote Web service site, and migrates back to the service requester site, together with the log database.

Step 10: The service requester creates a substitute component X_3' for the remote Web service at the local test environment, based upon the log database of the returned mobile agent and the information from the associated stub.

Step 11: The service requester starts the system integration test to test whether the system can tolerate the substitute component. The IPA technique is used to test the faulty propagation of the substitute component. First faulty data will be injected as input data to the substitute component. As a matter of fact, this fault injection has been performed at the time of designing the test data at Step 1.

Step 12: Faulty data will be injected as output data of the substitute component to its subsequent component Y_4.

Step 13: The service requester monitors the output data from the component X_3' and the subsequent components to test whether the errant action of the substitute component will be propagated to affect the whole system.

4. Discussions

In this section we will analyze and discuss our mobile agents-based approach, including both its merits and its limitations. Then we will envision some techniques and standards that are in demand in this field.

4.1 Merits

Here we evaluate whether our mobile agents-based approach achieves our goal stated at the beginning of this paper: to provide one possible solution to help select reliable Web services components. We examine whether our method solves the two research questions.

Question 1: *How to effectively and efficiently test remote Web services?*

By applying mobile agents technology to help Web services components selection, our mobile agents-based approach is more efficient by reducing the overhead incurred by invoking remote Web services. Service requesters can be released from being forced to remain the bindings with every candidate Web service. In addition, by dispatching multiple mobile agents to multiple candidate Web services simultaneously, the parallelism can be largely increased. Furthermore, by mobile agents performing testing at the Web services sites, we largely decrease the possibility of malicious attacks to every remote request. Therefore, utilizing mobile agents technology can make the selection process faster, safer, and more resource efficient.

In addition, by using assertions to test the vulnerability and functional quality of remote Web services, our mobile agents-based approach provides a dynamic strategy for service requesters to quickly identify the reliability of Web services. This strategy can also benefit the service providers by alleviating traffic.

Question 2: *How to test reliability of remote Web*

services?

By (1) constructing test data including normal data and corrupted data, and (2) setting up assertions according to the operational profiles of the service requester and the functional requirements of the desired component, our mobile agents-based 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 mobile 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, mobile agents can carry a matrix of test data to a remote Web service to test more "ilities".

Question 3: *How to test the interoperability of a remote Web service in a specific software environment?*

By perturbing the output data of a Web service gathered by mobile agents in a system integration test, our mobile agents-based 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 mobile agents-based 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.

4.2 Limitations

It should be noted that our mobile agents-based approach might not be applicable to some types of Web services. For example, considering a Web service such as flight tickets reservation, it is impractical to use mobile agents to test its quality. Under this kind of circumstances, the reputation of the specific Web service provider may be the main criterion for the decision.

In order to make our mobile agents-based approach more effective and feasible, we need several code generation tools. First, the mobile agent code needs to be automatically generated, or partially automatically generated. Second, we need a test data generator that can not only generate normal test data according to expected operational profiles, but also create corrupted test data that imitate anomalous events, including malicious attack events.

It should be noted that in order for our mobile agents-based approach to go alive, the authentication against incoming mobile agents from the Web service providers' sites have to be employed. The Web services' sites have to trust the mobile agents before letting them performing tests locally. Our current research does not address the solution to this issue and we will explore it in our future work.

Meanwhile, this paper only presents a high level model of our mobile agents-based approach. Several detailed techniques, such as how to integrate IPA with our Color Petri-Net based model WS-Net and how to perturb normal test data to imitate corrupted data, are not discussed in detail in this paper.

4.3 Further demand

In the future, more powerful Web services specification languages should allow service providers to publish their test suites with all the test data and corresponding results. In other words, the whole set or partial set of the log (i.e., the state of the remote Web service) that our mobile agents try to obtain can be gained from the published site. That being the case, the mobile agents will merely need to test upon the test data that are not found in the published test cases. Therefore, significant amount of efforts can be saved. Of course, the prerequisite of this approach is that the test cases published by the service providers should be trustworthy. The reputation of the service providers or some certification agencies can be of help on this issue.

Current publication and description languages for Web services (e.g., WSDL) do not provide the facilities to define the non-functional features of Web services, such as Quality of Service (QoS) features, constraints, Web service provider's reputation, etc. We envision that extensions will be added to the Web services publication and description languages.

5. Related Work

Voas and colleagues introduce an advanced fault injection technique called Interface Propagation Analysis (IPA) to test upon black-box like software systems [14]. IPA technique injects corrupted data to the input of a black-box system, and monitors the output of the system

to obtain knowledge of its fault tolerance. In this research, we apply the concept of IPA to test both the fault tolerance of Web services as stand-alone Web applications and the system integration interoperability of Web services serving as components in larger systems.

Kassab and Voas [19] propose to adopt the fault injection technique to fortify mobile agents. They inject faults into mobile agents to obtain higher observability. Comparing to their work, we adopt the fault injection technique for different purposes. Contract to their work that uses the fault injection technique to safeguard mobile agents, we equip mobile agents with fault injection stingers to poke at remote Web service sites to (1) test the fault tolerance of remote Web services, and (2) obtain full states of remote Web services so as to facilitate the local interoperability testing with remote Web services as components. In addition, their faulty data injected into mobile agents intend to fortify mobile agents themselves; thus, they are generated based upon the internal code of mobile agents. On the other hand, our faulty data injected into mobile agents intend to test remote Web services; thus, they are generated based upon the operational profiles of the local system and the interfaces of the remote Web services published in WSDL.

6. Conclusions and Future Work

In this paper we propose a mobile agent-based, fault injection-equipped, and assertion-oriented approach to help effectively and efficiently select appropriate Web services components to ensure the reliability of software system. Our future work will include: (1) constructing code generation tools for mobile agents and their test data, including both normal test data and corrupted data, (2) exploring more selection criteria to ensure more non-functional attributes, and (3) conducting more case studies.

Acknowledgement

The author deeply appreciates Dr. Jeff Voas for the initial inspirational discussion that directly leads to this research. The author also sincerely appreciates the anonymous reviewers for their insightful and helpful comments.

7. References

[1] C. Ferris and J. Farrell, "What Are Web Services?", *Communications of the ACM*, Jun. 2003. 46(6), pp. 31.

[2] P. Holland, "Building Web Services from Existing Application", *eAI Journal*, Sep. 2002, pp. 45-47.

[3] D.L. Parnas, A.J.V. Schouwen, and S.P. Kwan, "Evaluation of Safety-critical Software", *Communications of the ACM,* Jun. 1990. 33(6), pp. 636-648.

[4] P. Neumann, "Principled Assuredly Trustworthy Composable Architectures", emerging draft of the final report for DARPA's Composable High-Assurance Trustworthy Systems (CHATS) program, http://www.csl.sri.com/users/neumann/chats4.pdf.

[5] N. Gold, C. Knight, A. Mohan, and M. Munro, "Understanding Service-Oriented Software", *IEEE Software*, Mar./Apr. 2004, pp. 71-77.

[6] R. Han, V. Perret, and M. Naghshineh. "WebSplitter: a Unified XML Framework for Multi-Device Collaborative Web Browsing", *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work (CSCW)*, 2000, Philadelphia, PA, USA. pp. 221-230.

[7] J. Roy and A. Ramanujan, "Understanding Web Services", *IEEE IT Professional*, Nov. 2001, pp. 69-73.

[8] Simple Object Access Protocol (SOAP) 1.1, World Wide Web Consortium (W3C), May 2000, http://www.w3.org/TR/SOAP.

[9] http://www.w3.org/TR/wsdl.

[10] http://www.uddi.org.

[11] K. Rothermel and e. R. Popescu-Zeletin, "Mobile Agents", *Lecture Notes in Computer Science Series*, 1997, Vol. 1219.

[12] A. Pham and A. Karmouch, "*Mobile Software Agents: An Overview", IEEE Communications magazine*, Jul. 1998. 36(7), pp. 26-37.

[13] D.B. Lange and M. Oshima, "Seven Good Reasons for Mobile Agents", *Communications of the ACM*, 1999, 42(3), pp. 88-89.

[14] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*, 1998, New York: John Wiley & Sons, ISBN 0-471-18381-4.

[15] J. Voas, "Certifying Off-The-Shelf Software Components", *IEEE Software*, Jun. 1998, pp. 53-57.

[16] J. Voas, F. Charron, and K. Miller, "Robust Software Interfaces: Can COTS-based Systems Be Trusted Without Them?", *Proceedings of the 15th International Conference on Computer Safety, Reliability, and Security, Springer-Verlag*. Oct., 1996.

[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),* Jul. 6-9, 2004, San Diego, CA, USA, pp. 420-427.

[18] B.A. Mueller and D.O. Hoshizaki, "Using Semantic Assertion Technology to Test Application Software", *Proceedings of Quality Week*, May, 1994.

[19] L. Kassab and J. Voas, "Agent Trustworthiness", *Proceedings of 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations, Springer*. Jul. 20-24, 1998, pp. 121-133.