Wireless Pers Commun (2011) 56:147–172
DOI 10.1007/s11277-009-9879-9

A Mobile Agent-Based Tool Supporting Web Services Testing

Jia Zhang

Published online: 7 January 2010

Abstract The Web services technology has received significant momentum in recent years, because it allows people to easily utilize and integrate existing software applications to rapidly create new business services. However, how to ensure the trustworthiness of a Web services-oriented system remains a big challenge. One critical issue is how to test a Web service in an effective and efficient manner. In this paper, we report our design and development of a novel mobile agent-based method oriented to Web services testing. We seamlessly integrate an existing testing tool (HP LoadRunner) with a mobile agent technology (IBM Aglet) to build a practical environment for testing Web services. We also report an automatic test case generation algorithm, which analyzes published Web service interfaces and creates boundary value-based, fault injection-equipped test cases.

Keywords Web services testing · Mobile agent · Testing platform

1 Introduction

A Web service refers to a programmable Web application with a standard interface and is universally accessible using standard network protocols. Its interoperability provides users a practical way to quickly create a new service by combining existing services. It is one central reason why this emerging technology has caught significant momentum from both academia and industry in the recent years.

However, the wide adoption of the Web services technology has encountered a bottleneck, because it remains a big challenge as how to ensure the trustworthiness of a Web services-

This paper is a significant extension to its conference version: Jia Zhang and Di Xu, "A Mobile Agent-Supported Web Services Testing Platform", Proceedings of The 3rd International Symposium on Trustworthiness, Reliability and services in Ubiquitous, and Sensor Networks (TRUST 2008), Shanghai, China, Dec. 17–20, 2008, pp. 637–644.

J. Zhang (✉)
Department of Computer Science, Northern Illinois University, DeKalb, IL 60115, USA
e-mail: jiazhang@cs.niu.edu

based system. To tackle this challenge, one key prerequisite is how to test a Web service, especially from client side. The reason is obvious: eventually a client will not really trust a system unless he/she can fully test it.

The last 50 years of software development history has established an independent research branch called *software testing,* which contains a wealth of theories, technologies, methodologies, and tools to guide the verification process of a software product. In the recent years, significant efforts have been conducted to adapt the traditional testing technologies toward testing Web applications remotely over the Internet [1]. A number of tools have been developed. Among them, HP's LoadRunner [2] is a well-known performance and load testing software for examining system behaviors. In addition to testing traditional software products deployed in a local environment, LoadRunner has added some features to automate Web application testing.

However, these Web application-oriented testing technologies may not be suitable for measuring and testing Web services. The model of Web services poses significant challenges on Web application testing due to its unique nature and properties. First, unlike screen-centric Web applications, Web services are Web components only accessible via interfaces published in standard Web services-specific interface definition languages (e.g., WSDL) and accessible via standard network protocols (e.g., SOAP). Therefore, how to design test cases for a Web service using its limited information exposed remains a challenge. Second, one major goal of adopting the Web services technology is to dynamically compose existing Web services as components to quickly construct a new service. Therefore, Web services testing usually imply that multiple service components have to be tested in an efficient and lightweight (i.e., without maintaining dedicate network connections throughout a testing process) manner. Third, how to mitigate the overhead caused by the Web services-specific transport protocols (e.g., SOAP) deserves special attention [3].

The issue of Web services testing has caught significant attention in the recent years. For example, the latest version of LoadRunner (9.0) is also moving toward testing Web services. However, the current version of LoadRunner can only test single Web services with all test cases prepared ahead of time. Meanwhile, it requires to keeping the network connection throughout the entire testing process, which is not efficient and may not be even possible in the wireless network. Our previous work applies the mobile agent technology on Web services testing for higher performance and reliability [4]. A mobile agent is a composition of computer software and data that are able to travel from one computer to another on the network, and autonomously and automatically continue their execution on the destination computer [5–7]. IBM Tokyo Research Lab publishes Aglets Software Development Kit (ASDK) [8] that is an open source mobile agent development package written in Java. Our mobile agent-based Web services testing method can also handle testing multiple Web services with predefined testing scenarios and testing paths. However, our previous work requires that users write test code and embed them into mobile agents before they migrate.

This research is a continuous effort, aiming to build a practical method for Web services testing. Our contribution is two-fold. First, we seamlessly integrate an existing automatic software testing tool and the mobile agent technique to create an effective and efficient testing platform to facilitate Web services testing processes. In this project, we integrate IBM Aglet technology and HP LoadRunner as a proof of concept. On one hand, we apply the mobile agent technology to enhance the ability of a popular software testing tool (LoadRunner) on Web services testing; on the other hand, we integrate our mobile agent-based Web services testing approach with an existing tool to make it more practical and powerful. Second, we propose an automatic test case generation algorithm, which automatically analyzes published

Web service interfaces and creates boundary value-based, fault injection-equipped test cases. Instead of proving that a Web service under testing meets all requirements, we have chosen an alternative way to prove that a service cannot satisfy the requirements. Here a Web service refers to either a standalone Web service or a composite Web service comprising multiple individual Web services as components.

The remainder of the paper is organized as follows. In Sect. 2, we discuss related work. In Sect. 3, we briefly summarize our studies on LoadRunner and Aglet and explain the technical challenges of integrating the two techniques. In Sect. 4, we present the design of our testing platform. In Sect. 5, we present our algorithm of automatic test case generation. In Sect. 6, we present the implementation details of our testing platform and test case generation algorithm. In Sect. 7, we discuss our experiments. In Sect. 8, we make conclusions.

2 Related Work

2.1 Web Services Testing

Tsai et al. [9] believe that all Web services parties (including service providers, service brokers, and service requestors) must collaborate to perform Web services testing [9,10]. The essential technique of their collaborative group testing approach is to construct trustworthy service brokers and utilize distributed agents to rank and vote for appropriate services based upon service histories. In contrast with their work focusing on constructing a testing mechanism at service brokers to ensure quality of services at their check-in time, our research is a user-centric approach focusing on building a practical testing platform to help service requestors test published Web services.

Bai et al. [11] present an adaptive Web services testing framework that continuously learns and selects test cases based on testing results. Their another work presents a method of automatically generating test cases based on Web services interfaces in WSDL [12]. However, their WSDL-based test case generation stays at a high level. By contrast, we thoroughly examined XML built-in primitive types, studied their constraining facets, and extracted both boundary values and perturbed data to enhance test case generation. In addition, we built a practical test platform to enable automatic test case generation.

Ramsokul et al. [13] present a test bed to help service developers test the functional aspects of their implementations. In contrast with their work focusing on helping service developers, our work focuses on helping service requestors test multiple service candidates effectively and efficiently.

Maximilien and Singh propose to adopt dedicated agents to gather, store, aggregate, and share quality of service data to help dynamic service selection [14]. In contrast with their work focusing on designing the internal functions of agents to interact with Web services for useful information, we focus on using the mobile agent technology to enhance existing testing tools.

Our previous work [4] proposes a mobile agent-based approach to select Web service components. As a continuous effort, this paper reports how we developed a practical testing platform integrating existing widely used testing tools and the mobile agent technology: IBM's Aglet and HP's LoadRunner. Our another previous work [15] presents our preliminary study on how to find faulty data based on XML built-in primitive types for testing fault tolerance of Web services. This research project extends our previous work and studies generic Web services test case generation based on the entire set of XML built-in types.

Offutt and Xu propose to adopt data perturbation technique to generate test cases of testing message communications between pairs of Web services [16]. Their data perturbation includes two approaches: data value perturbation modifies values according to the data types specified by Web services; interaction perturbation tests on RPC communication and data communication. In contrast with their research using machine-related boundary values as data perturbation strategy (e.g., the largest number for a double data type), our research proposes a much finer-grained strategy to find boundary values based on the constraining facets and XML schema-referenced data type standards.

Researchers have constructed several formal models for describing XML documents [16,17]. Their works generally build a regular tree grammar to facilitate formal representation and derivation of XML documents. Compared to their works, we established a regular tree grammar for WSDL documents. Although a WSDL document is in fact an XML document, our purpose is to construct a model for WSDL document to facilitate automatic test case generation out of WSDL specifications.

2.2 Mobile Agents Research

Voas and McGraw introduce an advanced fault injection technique called Interface Propagation Analysis (IPA) to test on black-box-like software systems [18]. The 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. However, the IPA technique is rather a high-level guideline than a concrete methodology. In the context of Web services, we applied the IPA concept and established an automatic test case generation approach.

Kassab and Voas [19] propose to inject faults into mobile agents to obtain higher observability. Their faulty data intend to fortify mobile agents themselves; thus, they are generated based on the internal code of mobile agents. By contrast, our faulty data injected into mobile agents intend to test remote Web services; thus, they are generated based on the interfaces of the remote Web services published in WSDL.

Security issue has been a big concern for the mobile agent technology [20–23]. Thus, a significant amount of research efforts have been reported in the literature [24]. Trusted managers [25] and sliding encryption (asymmetric encryption) [26], as two examples among many proposed solutions, can be used to protect not only mobile agents but also agent platforms (migration hosts). Since the focus of our research is to apply the mobile agent technology to build a practical environment for testing Web services, we adopt a *one-time proxy signature* method [27] for both mobile agents and service provider sites.

Several mobile agent platforms have been developed. Among them, IBM's Aglet [8] technology provides an open-source Java-based package for developers to embed customized code. In this research, we integrate Aglet with HP's LoadRunner to explore how to utilize Aglet to test Web services.

3 Challenges of Integrating LoadRunner and Aglet

To build a practical Web services testing platform, we explore a way to leverage the power of the existing testing tools and the mobile agent technology. Without losing generality, we examine HP's LoadRunner [2] and IBM's Aglet [8].

3.1 Study on LoadRunner

Among a number of existing Web application testing tools, HP's LoadRunner [2] is a well-known performance and load testing software for examining system behaviors. It can simulate virtual users for stress testing, and collect the results returned from the server side during the tests. We chose LoadRunner in our project, because it is widely used and commonly recognized as a leading software package in software testing area. In addition, LoadRunner is also moving toward testing Web services.

Our experiments show that the current version of LoadRunner has some limitations to be used for Web services testing. First, LoadRunner cannot recognize malicious or wrong results returned from Web services. As long as an SOAP message is returned, LoadRunner will consider it as a valid result, even if it is not. Furthermore, the SOAP message will not be recorded. As a result, users cannot verify testing results even if they desire to do so. Second, LoadRunner cannot parse returned XML messages from a server to an expected form. Third, LoadRunner can only test a single Web service by maintaining an HTTP link during the entire testing process, which is obviously inefficient and may cause a significant amount of network traffic.

We further studied the Service Level Agreements (SLAs) specifications of LoadRunner to explore how they can be applied for Web services testing. We found two categories of SLA status parameters useful: the first set is determined over a predefined timeline (Average Transaction Response Time or Errors per Second); the second is determined over an entire testing period (Total Hits per run, Average His per run, Total Throughput, and Average Throughput). Users can further conFig. load criteria (e.g., Running Vusers, Throughput, Hit per Second, Transaction per Second, and Passed Transaction per Second) and operators (e.g., Less than, Greater than or Equal to, and Between).

We also found that LoadRunner provides a set of built-in product-level facilities. For example, it can automatically record the action of an XML transaction and an HTTP transaction. We adopt the action "Add Transactions" to create a test scenario, by specifying the beginning and the end of the test and the name of the test. We focus on testing two types of load. The first one is "iterations," meaning that a single user repeats a test process for multiple times. The second one is "concurrent users," meaning that multiple users perform the same test process synchronously. The two load types can be combined to simulate a real world scenario, where multiple users consume the same service at the same time.

3.2 Study on Aglet

A mobile agent refers to a composition of computer software and data that are able to travel from one computer to another over a network, and autonomously and automatically continue their execution on the destination computer. The ability of traveling allows mobile agents to move to a Web service's site to interact with it directly [5–7]. This implies that using the mobile agent technology to test a Web service may significantly increase efficiency.

There exist several mobile agent implementations. In this research project, we decide to adopt IBM's Aglet technology to build mobile agents, mainly due to our familiarity with the technique. In addition, Aglet package is written in Java, which is compatible with LoadRunner that is also written in Java. Furthermore, Aglet is an open-source package that allows us to embed code for our study.

The foundation of Aglet technology is Aglets Software Development Kit (ASDK), which was originally developed at IBM Tokyo Research Lab and now is an open-source project. An aglet refers to a mobile agent written in Java language based on ASDK. In other words, an aglet is a Java object that can move from one host to another on the Internet. That is, an aglet executing on one host can halt its execution, dispatch itself to another host through the Internet, and resume its execution there. During its trip, an aglet carries code as well as data. The security implementations in Java Virtual Machine (JVM) make it safe for remote sites to host and run aglets.

ASDK includes a complete Java aglet platform, with a standalone server called *Tahiti*, and a set of libraries that allow developers to build aglets and to embed aglets in their applications. All aglets are hosted by a Tahiti server, which provides an environment for aglets to execute, and provides a GUI for users to control all aglets that either are created locally or migrate from a remote host.

3.3 Technical Challenges of Integrating LoadRunner and Aglet

To integrate LoadRunner and Aglet to build a Web services-oriented testing platform, we met four significant technical challenges: Java version issue, communication issue between Aglet and LoadRunner, testing platform design issue, and security issue.

Without losing generality, the integrated development environment (IDE) we use to support Web services is Sun's NetBeans [28], which is widely used to develop Web applications and Web services. We found that Aglet does not work with the current version of NetBeans. Our investigation revealed that the issue results from JDK version incompatibility. Aglet is built on ASDK 2.0.2 that uses JDK 1.4.1; Java Platform, Enterprise Edition (J2EE, the major Java support for Web applications) requires support from JDK 1.5.0 or later.

When we tried to integrate LoadRunner with Aglet, the biggest challenge we encountered was how to make LoadRunner cooperate with a Tahiti server. Our experiments show that LoadRunner cannot recognize any action performed at a Tahiti server. Thus, LoadRunner cannot act accordingly (e.g., generate test scripts according to events).

Assuming that we could solve the previous two issues, our third challenge is to design a practical testing platform to automate the testing process of a Web service, leveraging LoadRunner and Aglet.

Finally, the security issue requires special attention. The mobile agent technology requires that agents migrate to a remote site and execute there. One might question that some sites may not allow mobile agents to run on their local sites. This argument is definitely valid under many circumstances. However, under certain cases, this bar may not be a strict requirement. For example, a bioinformatics problem may need to gather information from a series of distributed data sources. In this situation, dispatching a mobile agent to visit the list of data sources in a predefined order is a feasible and preferable solution. In addition, Java-based mobile agent technology (e.g., Aglet) only allows mobile agents to execute on Tahiti servers inside of JVMs, whose sandbox feature provides a layer of security protection. Furthermore, a Web service may provide a platform on a testing server to host mobile agents. This setting can add another layer of security control. As a result, in this paper, we learned from some existing methods [24–27] to eliminate security issues; and our mobile agent-based testing platform is meant for appropriate domains and situations.

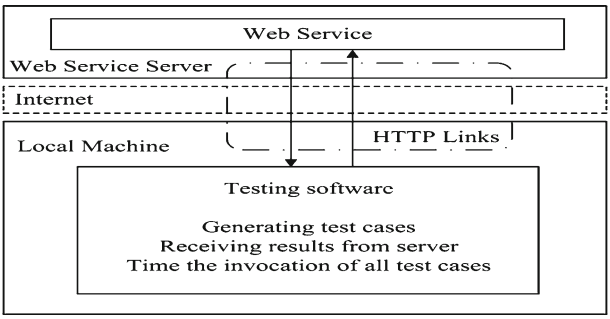

Fig. 1 Traditional Web services testing

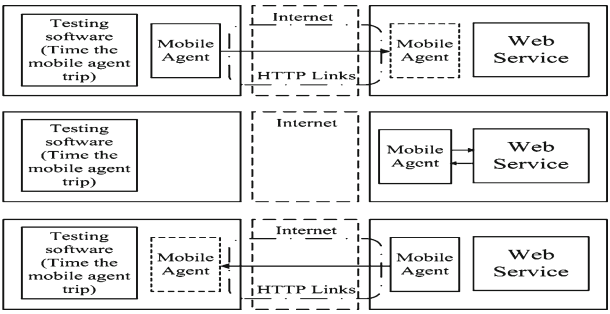

Fig. 2 Mobile agent method

4 Platform Design

4.1 Traditional Method

The testing protocol between LoadRunner and a Web service is shown in Fig. 1. LoadRunner running at a local host prepares test cases, starts a timer, opens and remains an HTTP link between the local host and the remote Web service, sends the test cases in serial to the remote Web service, and receives test results. After all test cases are processed, the HTTP link is terminated.

One major drawback of this model is that HTTP links have to be maintained open until all test cases are processed. In addition, every test case execution implies a pair of SOAP request message and SOAP response message. Therefore, this method may generate a significant amount of network traffic.

4.2 Mobile Agent-Based Method

Figure 2 shows how to apply mobile agents to test a Web service. A mobile agent is created at a local host carrying test cases. An HTTP link is created between the local host and the remote Web service and remained until the mobile agent travels to the Web service side. The mobile agent migrates to the remote site and invokes the carried test cases and collects testing results. Another HTTP link is then created, and the mobile agent migrates back to the local host carrying all testing results.

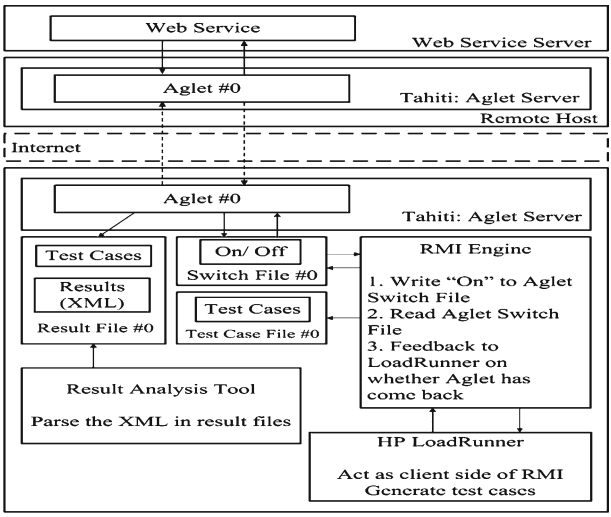

Fig. 3 RMI approach

In contrast with the traditional method where HTTP links remain open during the entire test scenario, the mobile agent-based method only keeps HTTP links open when mobile agents migrate. Meanwhile, all testing processes are locally conducted by mobile agents at the Web service site, instead of remotely as in the traditional method. Therefore, when the number of test cases becomes significant, the mobile agent-based method may greatly reduce the total testing time and network traffic.

4.3 Mobile Agent-Based Method with RMI

Our goal is to seamlessly integrate LoadRunner with Aglet. Unfortunately, we found that LoadRunner and Aglet cannot directly interact with each other. However, our experiments discovered that LoadRunner can record client-side actions of Java Remote Method Invocation (RMI) [29], and Aglet can operate on files. Thus, we built an RMI engine to bridge the gap between LoadRunner and Aglet, together with a switch file to control aglets in a Tahiti server. The key idea is illustrated in Fig. 3.

LoadRunner acts as the client side of RMI, and Aglet as the server side. As shown in Fig. 3, an aglet stays at the waiting mode checking the status of the aglet switch file. To start the process, LoadRunner sends a request including all test cases to the RMI engine. Upon receiving the request, the RMI engine writes all test cases into the test case file, writes an "On" into the aglet switch file, and then switches to waiting mode. Upon reading the signal, the aglet reads the test case file, loads itself with the test cases, and then starts its journey to the remote Web service. The returned aglet writes all test cases and results into a file, and then writes an "Off" into the aglet switch file to indicate its return. Receiving the signal, the RMI engine informs LoadRunner. A separate analysis program can be invoked to analyze the test results carried back by the aglet.

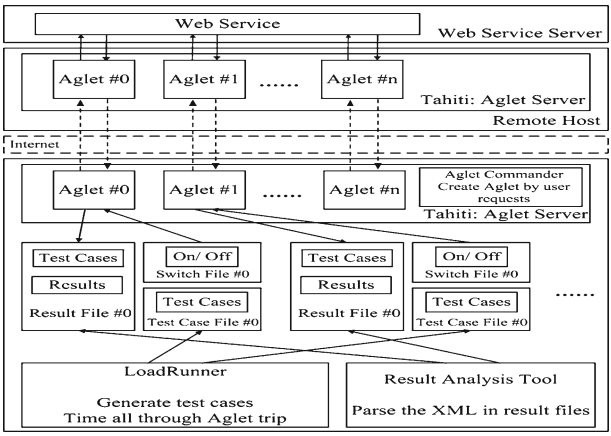

Fig. 4 Overview of testing platform

4.4 Mobile Agent-Based Test Platform

While employing the RMI technique is heavy weighted, we continued to explore a lighter-weight approach. Through experiments, we further found that LoadRunner can write into files as what a Java class can do. We managed to edit an empty test script file created by LoadRunner, without letting LoadRunner record our actions. This means LoadRunner can control a switch file to indirectly control aglet actions.

We designed and developed a mobile agent-based testing platform, as shown in Fig. 4. Our approach can support multiple mobile agents for a higher efficiency. A set of aglet-based mobile agents are created in a Tahiti server and stay alive. LoadRunner creates a set of test cases and writes them into the test case file. Then it turns on the switch file. Afterwards, the mobile agents each reads the test case file to equip itself, and then migrates to the remote Web services' sites to perform the tests. Finally, the agents carry back the results and write to the test case results file, and then turn off the switch file to inform LoadRunner of the completeness of the test cases.

As shown in Fig. 4, we designed a Result Analysis Tool to examine the text-based results that agents carry back for evaluation and analysis. After the return of each aglet, a result file is created for future evaluation of the test performed by this aglet at the server. The file contains two parts: the test case and result part and the test overview part. Below is a sample of test case:

Test Case #2
Lat: 43.404
Lon: −94.772
Report Start Time: 040101000000
Report End Time: 120419000000
Result:
<Fake SOAP String>

The test case contains some overview information about the test case that the aglet performs at the remote host: the ID of the test case, the value of the test case, time to start, and time to return. Below is a sample of test case result generated by our testing tool:

Time for Aglet Preparing Test Cases: 38ms
Time for Aglet Transferring to Server: 82ms
Total Test Cases Invoked: 50
Total Exceptions Occurred: 0
Total Invocation Time: 1953ms
Average Invocation Time of One Test Case: 39ms
Time for Aglet Transferring to Client: 471ms

5 Automatic Test Case Generation

5.1 Focus and Assumptions

Our platform allows mobile agents to carry test cases to remote Web services sites to conduct local testing on behalf of service requestors through the corresponding Intranet. The next question is what test cases a mobile agent should carry and how many it should carry. Because it is typical that a service requestor faces many service candidates with similar or the same functions, it is important to first remove unqualified service candidates to eliminate the selection pool. Therefore, instead of proving that a Web service is appropriate and could be selected, our strategy goes toward the opposite direction, aiming to create test cases focusing on breaking a Web service candidate and eliminating it from the candidate list. The output of our method is a much smaller pool of service candidates for further investigation.

To enhance performance, a mobile agent also carries assertion conditions. It can decide to stop testing as soon as it considers the Web service under test does not meet the lower-threshold criteria defined by the assertion conditions. Before discussing our test case generation approach, we will list our seven assumptions for simplicity reasons.

Assumption 1 The prerequisite of our work is that the functional requirements of service selection have been predefined, also a set of service candidates have been identified through some services discovery techniques.

Assumption 2 Test cases can be generated either by service requestors (agent creators) or mobile agents. In theory, the latter option may further improve performance by each mobile agent generating test cases in parallel. For simplicity reason, in our current research, we rely on service requestors to create test cases, and mobile agents only carry test cases. We plan to explore the latter option in our future work.

Assumption 3 Test case generation mainly depends on test goals. In this research, we focus on automatically generating test cases to break Web services under test. More specific test case generation can be aggregated by the test cases created for individual purposes such as reliability, security, safety, availability, and fault tolerance.

Assumption 4 We assume that service requestors decide assertion conditions mainly based on predefined service selection requirements and operational profiles (i.e., specific usage scenarios). In addition, security-related assertion conditions should also be considered. The detailed decision strategies and algorithms are not included in the scope of this paper.

Assumption 5 Service requestor-oriented operational profiles are typically application -specific and context-related. For simplicity reason, in this paper we will only consider test case generation based on published interfaces. Operational profiles-aware test case generation will be explored in our future research.

Assumption 6 Each Web service contains one operation. In reality, a Web service may contain multiple operations. We only consider one operation of a Web service for simplicity reason. Our method of generating test cases over one operation can be easily extended to Web services that comprise multiple operations.

Assumption 7 Since the major goal of this paper is automatic test case generation, we consider standalone Web services instead of composite Web services comprising multiple service components.

5.2 Boundary Value Deduction

According to Assumption 5, test cases are designed based on published Web service interfaces. The *ad hoc* industry standard for describing the interface of a Web service is Web Services Definition Language (WSDL). Therefore, our test case design is based on the limited information exposed by the WSDL documents associated with the Web services to be tested.

WSDL is "an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information" [30]. In more detail, a WSDL document includes a set of operations with required input and output parameters organized in input and out messages, respectively. Recall that our main goal is to design test cases to quickly prove a Web service as unqualified. Our basic strategy is to design test cases based on the boundary values of each parameter defined in the input message of the operation of the Web service. The challenge thus becomes how to effectively find boundary values for each input parameter.

According to WSDL specification, each parameter must be an XML-compatible data type, either an XML built-in type or a user-defined compound type based on built-in types. XML built-in types include built-in simple types and built-in complex types. The former can be in turn divided into built-in primitive types and built-in derived types; the latter is defined in terms of the former by unioning value spaces and lexical spaces [31]. In summary, any XML-compatible data types are built on top of built-in types. Therefore, we examine their categorizations and features.

W3C specification of the XML Schema (metadata) defines 19 built-in primitive types and 27 built-in derived types, summarized in Table 1 [31]. Every type is associated with a set of constraining facets, each restricting an aspect of the value space of the type (e.g., minimum value and maximum value). For example, the type *string* has six associated constraining facets: length, minLength, maxLength, pattern, enumeration, and whiteSpace.

Our previous work studies built-in primitive types [15]. We extend our previous study to built-in derived types and summarize the constraining facets for all XML built-in types in Table 1. The upper section contains the constraining facets for built-in primitive types; the lower, grayed section contains the constraining facets for built-in derived types. The reason why we create this table is that we use it as a guideline to analyze Web service interfaces and find boundary values as test cases, which details will be described later. Here, we only list the constraining facets relating to boundary value-based test case generation. For example, maxScale for the type String is not included in the table. In addition, pattern facet may guide to generate test cases based on regular expressions specified. It is not included in Table 1 and will be studied in our future research.

As shown in Table 1, a cell marked with "$\surd$" means that a constraining facet can be defined for the corresponding built-in type (but not required). For example, the column of "L" for

Table 1 Summary of constraining facets for XML built-in types

Datatype	L	minL	maxL	wSp	enum	maxI	maxE	minI	minE
String	√	√	√		√				
Boolean									
Float					√	√	√	√	√
Double					√	√	√	√	√
Decimal					√	√	√	√	√
PrecisionDecimal					√	√	√	√	√
Duration					√	√	√	√	√
DateTime					√	√	√	√	√
Time					√	√	√	√	√
Date					√	√	√	√	√
gYearMonth					√	√	√	√	√
gYear					√	√	√	√	√
gMonthDay					√	√	√	√	√
gDay					√	√	√	√	√
gMonth					√	√	√	√	√
HexBinary		√	√		√				
Base64Binary		√	√		√				
AnyURI		√	√		√				
QName		√	√		√				
NOTATION		√	√		√				
NormalizedString		√	√		√				
Token		√	√		√				
Language		√	√		√				
IDREFS		√	√	√	√				
ENTITIES		√	√	√	√				
NMTOKEN		√	√		√				
NMTOKENS		√	√	√	√				
Name		√	√		√				
NCName		√	√		√				
ID		√	√		√				
IDREF		√	√		√				
ENTITY		√	√		√				
Integer					√	√	√	√	√
NonPositiveInteger					√	√	√	√	√
NegativeInteger					√	√	√	√	√
Long					√	√	√	√	√
Int					√	√	√	√	√
Short					√	√	√	√	√
Byte					√	√	√	√	√
NonNegativeInteger					√	√	√	√	√

Table 1 continued

Datatype	L	minL	maxL	wSp	enum	maxI	maxE	minI	minE
UnsignedLong					√	√	√	√	√
UnsignedInt					√	√	√	√	√
UnsignedShort					√	√	√	√	√
UnsignedByte					√	√	√	√	√
PositiveInteger					√	√	√	√	√
YearMonthDuration					√	√	√	√	√
DayTimeDuration					√	√	√	√	√

l length, *minL* minLength, *maxL* maxLength, *wSp* whiteSpace, *enum* enumeration, *maxI* maxInclusive, *maxE* maxExclusive, *minI* minInclusive, *minE* minExclusive

string in the first row is marked, which means that a user can define a string limit for an instance of the type *string*. A cell left blank means that the constraining facet is not allowed to be defined for the corresponding built-in type (e.g., maxInclusive for the type *string* in the first row). Certain constraining facets for some built-in types have been predefined to some default values, which can be overwritten by users. For example, the default maxInclusive value for the type *nonPositiveInteger* is predefined as "0." For simplicity reason, predefined constraining facet values are not included in Table 1 that is already crowded.

As shown in Table 1, 9 types of constraining facets are related: (1) *length* (L): the number of units of length based on data types, (2) *minLength* (minL): the minimum number of units of length, (3) *maxLength* (maxL): the maximum number of units of length, (4) *whitespace* (wSp): space, tab, line feed, and carriage return, (5) *enumeration* (enum): a set of values, (6) *maxInclusive* (maxI): inclusive upper bound, (7) *maxExclusive*: exclusive upper bound, (8) *minInclusive* (minI): inclusive lower bound, and (9) *minExclusive* (minE): exclusive lower bound. Complete constraining facet definitions can be found in W3C XML Schema [31].

We divide the 9 constraining facets into four categories based on how they can be used to create boundary value-based test cases.

(1) Four constraining facets explicitly specify the boundary values to test: *maxInclusive*, *minInclusive*, *maxExclusive*, and *minExclusive*.
(2) One constraining facet explicitly defines the set of values to test: *enumeration*.
(3) Three constraining facets define the length of a testing argument: *length*, *minLength*, *maxLength*.
(4) *WhiteSpace* facet guides to generate test cases on spaces.

The boundary value deduction algorithm is shown in Fig. 5 in pseudo code. We use the constraining facets as guidelines to identify boundary values. For a WSDL document, we find its associated XML Schema definitions that may be either included in the corresponding WSDL definitions or linked through separate XSD files. For each input parameter in a WSDL operation, we search for its constraining facets from the corresponding XML schema definitions. For example, consider a WSDL input parameter defined as an XML data type *ID* with constraining facet of length: <length value = '16'>. A boundary value can be created as an ID using a 16-character string. The keywords for the nine constraining facets are used to search for the relative specifications. If a value is defined, the corresponding constraining facet name will be used as an attribute. For the above example, the keyword "length" can be used to search for the attribute "length" in the corresponding XSD specifications for the

```
for each input parameter from the defined input message {
   in WSDL files || XSD files,
   if (found constraining facet keyword) {
      if (type 1 || type 2 || type 4) obtain boundary values;
      if (type 3) obtain a set of boundary values;
   } else {
      if (number type) use IEEE standards to deduce boundary values;
      if (time type) use ISO standards to deduce boundary values;
   }
}
```

Fig. 5 Boundary value deduction algorithm

Table 2 Perturbation strategy to generate faulty data

Constraining facets	Perturbation strategy
Length	Value + 1, value − 1
MinLength	Value − 1
MaxLength	Value + 1
WhiteSpace	Null, tabs, space, multiple spaces
Enumeration	Values not included in the set
MaxInclusive	Value + 1
MaxExclusive	The value
MinInclusive	The value
MinExclusive	Value − 1

constraining facet *length* and its specified value "16." The first, second, and fourth categories explicitly define the boundary values that can be used. The third category specifies the length of test case values.

If a user-defined constraining facet value is not found, then we check whether there exists an XML predefined default value. For example, we will use the default maxInclusive value "0" for a parameter type *nonPositiveInteger*. If no default value is defined, we use IEEE standards [31] for number types and International Standards Organization (ISO) standards for time type, to identify an implicit constraint value. For example, IEEE standards define (2^128 – 1) as the implicit *maxInclusive* value for type *float*.

5.3 Faulty Data Design

The IPA technique suggests injecting faulty data to test fault tolerance of a software program [19]. We apply IPA to inject faulty data to break a Web service. We propose to perturb extracted boundary values to create faulty data test cases. Our approach is again based on XML schema constraining facets. Table 2 summaries our methods of creating faulty data based on each constraining facet included in Table 1, by perturbing the boundary values. We intend to explore a generic method toward automatic test case generation.

(1) *length*: Since it defines the exact number of the characters or digits to be used in an argument, two test cases are generated, one with (length+1) and one with (length-1). (2) *minLength*: Since it defines the minimum number of characters or digits to be used in an argument, one test case is generated with a smaller length of (minLength-1). (3) *maxLength*: Since it defines the maximum number of characters or digits to be used in an argument, one test case is generated with a larger length (maxLength+1). (4) *whitespace*: Several test cases

are generated with values *null*, a tab, and one or multiple white spaces. (5) *enumeration*: Since it defines the set of values to be used, one or more test cases are generated with selected values not included in the set defined. (6) *maxInclusive*: Since it defines the largest value that can be used, one test case is generated with the value of (maxInclusive+1). (7) *maxExclusive*: Since it defines the largest value that cannot be used, one test case is generated with the value of (*maxExclusive*). (8) *minExclusive*: Since it defines the smallest value that cannot be used, one test case is generated with the value of (minInclusive-1). (9) *minExinclusive*: Since it defines the smallest value that can be used, one test case is generated with the value of (*minInclusive*).

Note that both of our boundary value deduction and faulty data perturbation algorithms are based on constraining facets. Our strategy is as follows. After deducing a boundary value, perturb it to create one or more faulty test cases. A suite of test cases will then be obtained by aggregating all test cases.

Note that both of our boundary value deduction and faulty data perturbation algorithms are based on constraining facets. Our strategy is as follows. After deducing a boundary value, perturb it to create one or more faulty test cases. A suite of test cases will then be obtained by aggregating all test cases.

5.4 Automatic Test Case Generation

Without the ability to automatically generate test cases, our mobile agents-based method would be burdened with the responsibility of manually analyzing WSDL documents and creating test data, introducing the likelihood of errors and omissions. Therefore, we propose an automatic test case generation mechanism in our work.

Since WSDL documents are XML files, any XML parser, in theory, could be used to parse WSDL documents. Our previous research yields a Web application code generator WebGen [32]. We extend our tool in this work to parse WSDL documents and automatically generate test cases.

Figure 6 shows the workflow of our automatic test case generation process. After loading a WSDL document, our tool parses it into an XML tree. According to WSDL specifications, the tags <portType>, <function>, and <input> are sequentially searched in the generated XML tree. Any tree traversal algorithm can be used. We use post-order algorithm, where any node is visited after all of its children are visited.

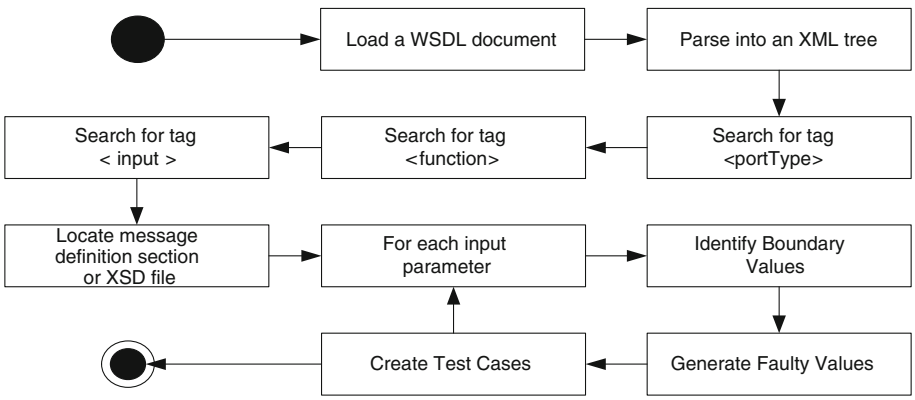

Fig. 6 Automatic test case generation workflow

For each operation, we record its name, and examines whether the operation requires an input message and an output message. WSDL defines four types of invocation patterns: one-way, request-response, solicit-response, and notification [30]. Depending on the invocation pattern, an operation may or may not have both input message and output message. For simplicity, we only consider the request-response pattern, implying that the operation contains both an input message and an output message. The other three methods can be dealt with in similar ways.

Using the name of the found input message, its schema definition can be found either in the same document (tree) or in a separate XSD document, which is in turn parsed into an XML tree. By iterating on each input parameter, its boundary values are identified and then perturbed into faulty data. In more detail, the tool identifies every parameter type of the input message. If a parameter type is an XSD complex data type, the tool uses the parameter name to search for the corresponding type definition and recursively navigates through each element to obtain the complete data structure of the parameter type. The tool also stores the gained information to relational database tables, for the purpose of reusability. Since a Web service is hosted by its own service provider, from the service requestor's perspective, the Web service might need to be retested at a later time even if it has been tested previously. As long as the WSDL definition of a tested Web service remains the same, its data structure will remain the same thus can be reused.

Finally, test cases are generated. Each test case contains a data state, which is a set of (name, value) pairs representing all input parameters and their test values. For an example, consider a data set below:

{(*name, "Jia"*), (*phone, 815-753-5947*), (*categoryId, 1*)}

This data set indicates that there are three input arguments in a test: a name, a phone number, and a category id. The name argument is a string value of "Jia"; the category id is an integer value of "1"; the phone argument is a string value of "*815-753-5947."*

5.5 Automatic SOAP Message Generation

A mobile agent carries generated test cases and migrates to the agent platform of a Web service. As we discussed in the last section, each test case comprises a set of (name, value) pairs representing the arguments for one test input. In this way, a mobile agent carries less data. On the other hand, Web services can only be accessed via SOAP messages. Therefore, our mobile agents have to carry intelligence to generate test SOAP messages after they migrate to the service providers' sites.

We utilize the generic code generator [32] from our previous research to realize the automatic SOAP message generation by mobile agents. For detailed description of our generic code generator please refer to [32]. In short, our generic code generator is capable of generating any code in any language based on given templates, data files, and configuration files. In this SOAP message generation task, data files are XML files comprising test cases, in the format as follows, where each test case contains one argument for *productid* as the XML schema type *string*.

```
<genericGenerator>
  <OBJ NAME="TestCase">
      <ATTR ATNAME="productid" ATYPE="String" UNIQ="NO"/>
    </OBJ>
</genericGenerator>
```

```
POST /InServicesComputing HTTP/1.1
Host: www.onlinemall.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 100

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Header>
    <m:Payment xmlns:m="http://www.onlinemall.org/prices/"
      soap:actor="http://www.onlinemall.org/appml/">
      soap:mustUnderstand="1">
        123
    </m:Payment>
</soap:Header>

  <soap:Body>
    <m:GetPrice xmlns:m="http://www.onlinemall.org/prices">
      <m:productid><_ATNAME></m:productid>
    </m:GetPrice>
</soap:Body>
</soap:Envelope>
```

Fig. 7 An example of SOAP message template

Recall that all test cases carried by a mobile agent to an agent platform aim to test the same functionality of a specific Web service. Therefore, all the SOAP messages to be generated should be in the same format. To minimize the mobile agent's on-site work, a SOAP message template is created by the service requestor. Figure 7 shows an example of a SOAP message template intending to test over a Web service for retrieval of product prices listed in a catalog.

As shown in Fig. 7, relating to a specific Web service, all the SOAP message headers and XML headers are fixed. The top section of the SOAP message template shows its binding to HTTP. The *Content-Type* header defines the MIME type (here application/soap+xml) for the message and the character encoding (here utf-8) used for the XML body. The *Content-Length* header specifies the number of bytes in the message body (here 100 bytes). In the *Body* element, such a SOAP message requests the price of a product by calling the function *GetPrice* with an argument *productid.* Note that the *m:GetPrice* and the *productid* elements are application-specific elements. They are not a part of the SOAP standard. Tag <_ATNAME> is to be replaced by the items listed in the data file (test cases).

In summary, a mobile agent carries a copy of our generic code generator, together with the SOAP message template and test cases. After migrating to the destination agent platform, the generic code generator automatically produces a set of SOAP messages by replacing the tags in the SOAP message template with the items defined in the data file.

5.6 Analysis of Test Results

For each test case, a mobile agent generates a SOAP input message to the testing Web service and receives a SOAP output message as a response. Upon analyzing the output message, a success or a failure is concluded. The mobile agent uses carried function to calculate the accumulated quality value after each test case. The value obtained is asserted against some predefined threshold. If the value goes below the predefined threshold, the Web service is concluded to have low quality and should be removed from the candidate list. The mobile agent thus sends back the assertion result, and terminates itself.

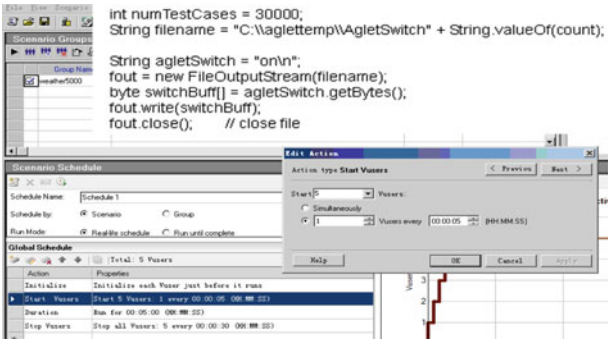

Fig. 8 Linkage of VUser and aglets

For each test case, the mobile agent analyzes the SOAP reply message and records the test result. If the quality value of a Web service is higher or equal to the predefined threshold once all test cases are conducted, then the original test cases and the corresponding test results will be packaged and sent back to the service requestor (agent dispatcher) for further testing.

6 System Implementation

6.1 Preparation of Aglet

We developed testing aglets to communicate with a Web service, record all testing results and include them in SOAP response messages, and then carry the results back. Users may analyze the results and decide whether they are correct or not.

LoadRunner can create multiple virtual users to simulate a load test. Each virtual user repeats the same action according to test scripts. To simulate load testing, we created a specific aglet manager to dispatch a number of aglets. Meanwhile, we used LoadRunner to record the round-trip travel time of each aglet. The key point is to assign unique identifiers to aglets to control and monitor their behaviors. In detail, we modified our test script to assign a unique ID to each of the virtual user created, so that each of them can only access one specific test case file and one specific aglet switch file.

Our key steps are illustrated in Fig. 8. We first specify the control script through LoadRunner Virtual User Generator. Switching to the edit mode, we define the number of test cases that need to be carried by each aglet according to the test plan. Second, we open LoadRunner Controller and specify the number of virtual users to be created. These two numbers have to be identical. A unique ID is thus assigned to each virtual user and to the aglet with which it communicates. Note that we have to specify the properties of the "Start Vuser" option in the "Global Schedule" section to "1 Vuser every 5 s." The reason is to ensure that each created virtual user is assigned a unique ID. If several virtual users start to be initialized at the same time or during a small time period, they may "share" an ID, which may cause problems in the later procedures.

As shown in Fig. 9, we developed a unique aglet named AgletCommander, which is created at the beginning of the preparation phase. Its major responsibility is to create test aglets according to user requirements, e.g., how many test aglets they want to create to simulate a load condition. Each aglet created by AgletCommander will obtain a unique ID. The IDs

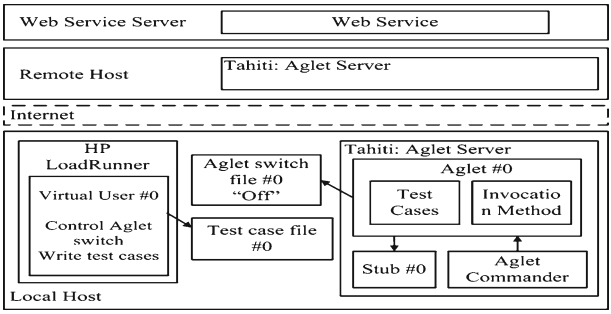

Fig. 9 AgletCommander

are identical to those of virtual users generated by LoadRunner. Thus, we ensure that each aglet simulates one virtual user. Meanwhile, running aglets periodically communicate with AgletCommander to inform that they are still alive.

6.2 Web Services-Oriented Agent Platform

Security concern may hinder the wide adoption of utilizing the mobile agent technology for Web services testing. One may argue that service providers would never allow service requestors to send mobile agents onto their sites to perform testing (invocations) for avoiding security attacks. However, this security issue is inherent for any mobile agent-based computing [20–23]. Nevertheless, the migratory characteristic of mobile agents promises significant merits for it to be applied to distributed computing environments [7]. Therefore, numerous research efforts have been conducted focusing on tackling the security challenges regarding mobile agents [24]. Trusted managers [25] and sliding encryption (asymmetric encryption) [26], as two examples among many proposed solutions, can be used to protect not only mobile agents but also agent platforms (migration hosts). The research achievements from the mobile agents community can be adopted to alleviate the security challenge of applying mobile agents to facilitate Web services testing.

In our research, we adopt a *one-time proxy signature* method [27] for both mobile agents and service provider sites. In more detail, a Web service provider generates a proxy key pair, and signs one and only one message with the pair. A service requestor, who is also the agent owner, then generates a message accordingly. This signature will be carried by the mobile agent to the Web service provider's site. Both the signatures of the service provider and the service requestor will be verified before testing begins. Detailed discussions about the method can be found in Kim et al. [27].

When exploring Web services-oriented agent platforms, consider another unique feature of Web services technology—a Web service is open to the Internet. This complete openness does add more security concerns for service providers to accept the mobile agents technique than it does for traditional network application providers. Therefore, we propose that a service provider establishes an agent platform on its local network to allow and enable mobile agent access. The physical machine(s) hosting Web services do not allow mobile agent access. Instead, as shown in Fig. 10, a dedicated agent platform is constructed on the Intranet of the service provider, assuming that the service provider decides to allow mobile agent access. In this way, it is fully up to a service provider to decide whether mobile agent access is allowed or not. The aforementioned concern can be alleviated.

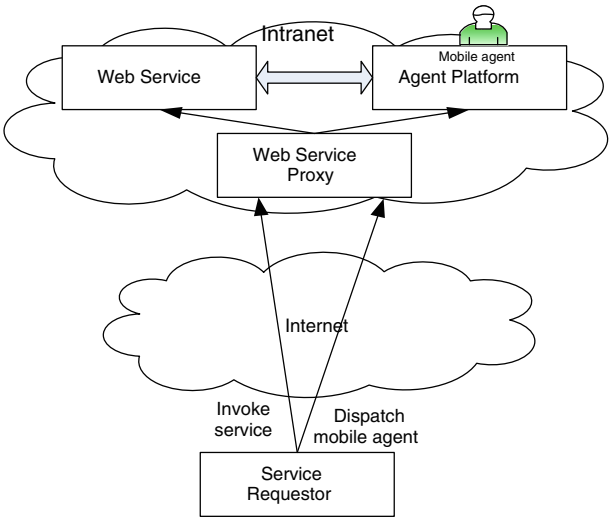

Fig. 10 Agent platform on service provider site

As shown in Fig. 10, the agent platform is transparent to service requestors. A Web service proxy acts as the dispatcher for incoming service requests. For a SOAP service request, the Web service dispatcher forwards it to the corresponding Web service. For an incoming mobile agent, the dispatcher directs the agent to the agent platform. Then the mobile agent can test the Web service from the agent platform through the Intranet.

The agent platform adds another layer of security protection for Web services. Authorization and authentication facilities can be equipped at the agent platform. Only after the mobile agent is launched on the agent platform, it can start to conduct tests on the Web service.

6.3 Internal Structure of Mobile Agent

Figure 11 illustrates the internal structure of an intelligent mobile agent. Five functional components are contained: (1) a control manager, (2) a test manager, (3) an assertion manager, (4) a SOAP message generator, and (5) a SOAP message interpreter. Three databases are identified: a test data repository, a knowledge database, and a log database.

The control manager can be considered as the Headquarters of the mobile agent: it controls all other modules. For example, it decides when to start testing, when to send back assertions, and when to terminate execution. The test manager loads test data from the test data database, and passes them to the SOAP message generator. The SOAP message generator component (using our generic code generator) generates SOAP request messages and sends to the Web service. The SOAP message interpreter module listens to SOAP response messages from the Web service, interprets the responses to obtain the test results, and passes the test results to the assertion generator. The assertion generator pairs the test data and corresponding test results together, utilizes the knowledge database to make assertion tests, and writes results to the log database. If a certain assertion is false, the assertion information may be passed to the SOAP message generator to create a SOAP message and send back to the service requestor. The test data repository may include predefined discrete test data set. The knowledge database provides information (including quality calculation function) to help the assertion manager

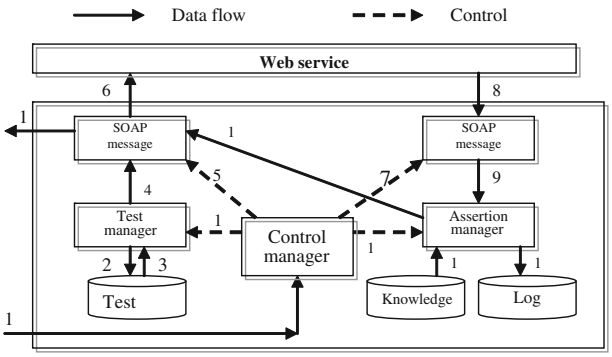

Fig. 11 Internal structure of a mobile agent

make decisions based on whether some predefined semantic assertions are satisfied or not. The log database stores pairs of test data and corresponding test results, as well as assertion information.

Figure 11 also shows the workflow of how a mobile agent works. The arrowed lines represent the information flow among the components: the solid directed lines represent the data flow, and the dotted directed lines represent the control flow. The numbers associated with each line indicate the execution order of the workflow. We will walk through the scenario as shown in Fig. 11 as follows:

Step 1: The control manager starts the mobile agent by sending control information to the test manager.
Step 2: The test manager queries the test data database to iterate through the test data set predefined.
Step 3: The test manager retrieves information from the test data database.
Step 4: The test manager generates test cases and passes them to the SOAP message generator.
Step 5: The control manager synchronizes the SOAP message generator to work.
Step 6: The SOAP message generator generates SOAP messages and sends them to the Web service.
Step 7: The control manager then synchronizes the SOAP message interpreter to listen to the responses from the Web service.
Step 8: The SOAP message interpreter catches the reply information from the Web service.
Step 9: The SOAP message interpreter analyzes the incoming SOAP response messages, obtains the test result data, and passes them to the assertion manager.
Step 10: The control manager synchronizes the assertion manager to work.
Step 11: The assertion manager retrieves information from the knowledge database to make decisions on assertions.
Step 12: The assertion manager stores in the log database pairs of test data and their corresponding test results, as well as the assertion results.
Step 13: If an assertion is false, the assertion manager sends the information to the SOAP message generator.
Step 14: The SOAP message generator generates a SOAP message and sends back to the service requestor.
Step 15: Based on the assertions received, the service requestor may send a message to the control manager to terminate the mobile agent's task. This message may control

the mobile agent to travel back to the service requestor with information gathered so far, or request that the mobile agent terminate itself. For example, if the service requestor suspects that the mobile agent has been maliciously attacked, its carried information becomes useless.

7 Experiments

7.1 Environmental Setup

JDK providing a JVM is the least requirement for using ASDK. In this project, the platform needs to support Web services. Therefore, JDK is not enough. We have to use J2EE package.

Java Platform, Enterprise Edition (J2EE) is the industry standard for developing Web applications based on Java. Java EE provides Web services, component model, management, and communication APIs for implementing service-oriented architecture (SOA) and next-generation Web applications including Web services [33]. It provides a framework for developing and deploying Web services on a Java platform.

We encountered an issue when we tried to integrate the IBM Aglet technique. The IDE we use to support Web services is Sun's NetBeans, whose J2EE component is with version 5. Aglet does not work with NetBeans. Our investigation found that the issue resulted from JDK version incompatibility. Aglet is built on ASDK 2.0.2, which uses JDK 1.4.1; J2EE requires support from JDK 1.5.0 or later. To solve this issue, we thoroughly compared the publish timelines of available JDKs and ASDKs, and then conducted a careful verification process. We found that J2EE 1.4 Update 3 is compatible with both systems. Thus, we adopted J2EE Update 3 as our Java platform, and bound both Aglet and NetBeans to the same platform.

Sun Java System Application Server (SJSAS) is bundled in the full version of J2EE. It is an open-source application server written in Java. It supports the deployment of Web services written in Java and provides basic security implementations using Java security implementations [34]. Note that SJSAS 8.2 is bundled in J2EE 1.4 Update 3. This is the main reason why we do not use other application servers, e.g., Glassfish (a newer version of SJSAS) or Apache Tomcat.

We set up a client/server environment. The server machine runs Windows XP with J2EE and Sun Java System Application Server installed. It acts as a service provider with a Web service deployed and published. Another identical machine equipped with the same platform and software packages acts as a client.

Aglet is Java-based technique, so that it is platform independent. We decided to choose Windows XP as the running platform, mainly because Aglet provides a GUI to control aglet servers on Windows XP, which is more user-friendly than using console-based interfaces. Since we develop all aglets using GUIs, the client machine has to run Windows XP, while it is not a requirement for the server machine. The server machine must have a Java run-time system installed with required libraries and run a Tahiti server.

Two other open-source software packages are used in our platform: NetBeans and Java Web Service Development Pack (JWSDK). NetBeans is a modular and standards-based IDE written in Java [28]. It supports the development of both Web services and Web service clients. It can directly communicate with the SJSAS installed on the same machine and help users deploy Web services without using administration console of SJSAS. JWSDK is an open-source software development package mainly for developing Web services and Web applications in Java [35]. It can be plugged into NetBeans 6.0.1. JWSDK provides two components for developing Web service clients to consume existing Web services: JAX-RPC

Table 3 Comparison of invocation time (in seconds)

# Of test cases	LR@L	AG@L	Δt
5,000	39	23.312	15.688
10,000	76	45.231	30.769
15,000	114	67.52	46.48
20,000	154	89.057	64.943
25,000	191	110.177	80.823
30,000	230	130.897	99.103
35,000	269	154.579	114.421
40,000	306	176.032	129.968
45,000	343	199.364	143.636
50,000	379	219.387	159.613

and JAX-WS. The former is specific for developing clients using synchronous calls to Web services [36], while the latter uses asynchronous calls [37].

7.2 Testing Platform

At the early stage of our project, we adopted a published Web service called "National Digital Forecast Database (NDFD) SOAP Web Service" (http://www.weather.gov/), provided by National Weather Service. The service responded correctly when we sent a small number of test cases. However, when the number of test cases increased, the service only returned error messages. This phenomenon is reasonable, as the Web service might take our test cases as service-level attacks. They may block the requests from an IP address if they receive a number of requests from the same IP address. Therefore, we developed and deployed our own weather Web service for testing purposes.

Without losing generality, this Web service provides one function: it receives the latitude and longitude of one location, the desired start time and end time of the report, and returns weather reports. For testing purpose, the internal execution time of the Web service is minimized. The testing Web service is hosted on a server machine and remains running. Testing programs run on a separate client machine. Both the server machine and client machine run Tahiti in order to enable the aglets' migrations. The server machine and the client machine communicate through our local network. Therefore, their communication time can be considered as a constant number.

On the local machine, we ran two sets of experiments, one using LoadRunner alone and one using our testing platform. Using the same testing scenario, we recorded time elapse for both LoadRunner method (LR@L) and aglet method (AG@L). To eliminate coincidence, we repeated our tests using different numbers of test cases, in the range of 5,000–50,000. Our testing results are summarized in Table 3.

When the number of test cases increases, our Aglet-based method significantly saves testing time, as shown in Fig. 12. By subtracting AG@L from LR@L (resulting a Δt as shown in Table 3), the value indicates how much time can be saved using the Aglet method. This value can also be used to analyze network condition if the value vibrates at some time point. Studying Table 3, we can find that with the number of test cases increased, our Aglet-based testing method can reduce the testing time by as much as 40%.

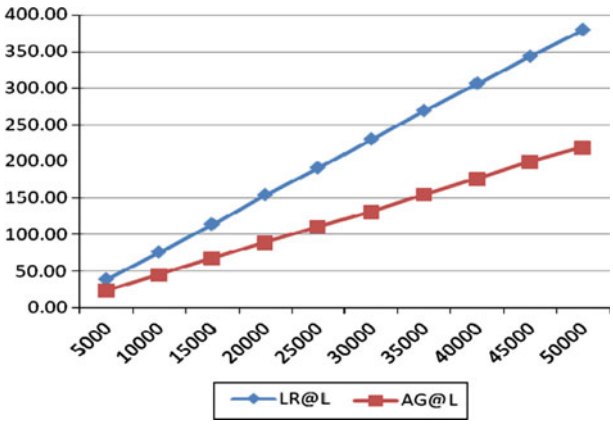

Fig. 12 Comparison of LoadRunner and our method

Our experiments prove that our approach has successfully integrated LoadRunner and Aglet technology to provide an effective and efficient testing platform oriented to Web services testing. The technical issues discussed in Sect. 3 are solved by our design and development.

8 Conclusions

In this paper, we reported our design and development of a Web services-oriented testing platform, by seamlessly applying the mobile agent technology to enhance an existing testing tool. An automatic test case generation approach is also reported based on boundary value deduction and faulty data injection.

Our work addresses the four technical challenges we identified at the beginning of our project. Our design and implementation of testing platform solves the first three issues (Java version issue, communication issue between Aglet and LoadRunner, and testing platform design issue). By building an agent platform with the assistance of a Web service proxy engine (discussed in Sect. 6.2), we address the security concern of applying the mobile agent technology for Web services testing.

Our research results can be applied to create a new generation of Web services testing tool. There are two ways to apply our results. One way is to extend existing testing tools to create a mobile agent-based tool. The other way is to adopt Service Oriented Architecture (SOA) concept to create a loosely coupled testing tool, meaning that equipping both testing tools and the mobile agent technique as two service components that can be assembled if so desired. In addition, applying the mobile agent technology to Web services testing eliminates significant network traffic, which sheds a light on conducting Web services testing using personal wireless computing devices.

Our current research work is based on HP LoadRunner and IBM Aglet. However, our approach is not limited to the two tools. Instead, our method can be extended to other testing tools and mobile agent packages.

Currently, our Web services testing platform adopts a synchronous mode. To further facilitate Web services testing in wireless Internet, we plan to build an asynchronous testing method on top of LoadRunner. In addition, our current automatic test case generation merely depends on published Web services interfaces. We plan to explore how to better decide an efficient number of test cases to be generated, by considering execution profiles.

We also plan to study the performance of our test case generation algorithm. Moreover, we plan to explore how to use personal wireless devices to control and manage Web services testing. Finally, we plan to build a Web site to host our testing platform on the Internet, so that we can publish our service to help users perform Web services testing.

Acknowledgment The author appreciates Di Xu from Northern Illinois University for his help of implementing a prototype of the Web services testing platform.

References

1. Elbaum, S., Rothermel, G., Karre, S., & Fisher, M. I. (2005). Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 187–202.
2. HP. *LoadRunner*. Accessed June 5, 2009, Available from http://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17%5E8_4000_100__.
3. Werner, C., Buschmann, C., & Fischer, S. (2005). WSDL-driven SOAP compression. *International Journal of Web Services Research, 2*(1), 14–35.
4. Zhang, J. (2005). A mobile agents-based approach to test reliability of web services. *International Journal of Web and Grid Services, 2*(1), 92–117.
5. Rothermel, K., & Popescu-Zeletin, E. R. (1997). *Mobile agents*. Lecture Notes in Computer Science Series, 1219.
6. Pham, A., & Karmouch, A. (1998). Mobile software agents: an overview. *IEEE Communications Magazine, 36*(7), 26–37.
7. Lange, D. B., & Oshima, M. (1999). Seven good reasons for mobile agents. *Communications of the ACM, 42*(3), 88–89.
8. IBM. *Aglets.* Accessed June 5, 2009, http://www.trl.ibm.com/aglets/.
9. Tsai, W. T., Chen, Y., Paul, R., Liao, N., & Huang, H. (2004). Cooperative and group testing in verification of dynamic composite web services. In *Proceedings of 28th annual international computer software and applications conference—workshops and fast abstracts—(COMPSAC)* (pp. 170–173), September.
10. Bai, X., Wang, Y., Dai, G., Tsai, W.-T., & Chen, Y. (2007). A framework for contract-based collaborative verification and validation of web services. In *Proceedings of the 10th international ACM SIGSOFT symposium on component-based software engineering (CBSE)* (pp. 258–273). Medford, MA, USA, July 9–11.
11. Bai, X., Chen, Y., & Shao, Z. (2007). Adaptive web services testing. In *Proceedings of IEEE 31st annual international computer software and applications conference (COMPSAC)* (pp. 233–236), July 24–27.
12. Bai, X., Dong, W., Tsai, W.-T., & Chen, Y. (2005). WSDL-based automatic test case generation for web services testing. In *Proceedings of IEEE international workshop on service-oriented system engineering (SOSE)* (pp. 215–220). Beijing, China, October 20–21.
13. Ramsokul, P., Sowmya, A., & Ramesh, S. (2007). A test bed for web services protocols. In *Proceedings of second international conference on internet and web applications and services (ICIW)* (pp. 16–21), May 13–19.
14. Maximilien, E. M., & Singh, M. P. (2004). A framework and ontology for dynamic web services selection. *IEEE Internet Computing, 8*(5), 84–93.
15. Zhang, J., & Qiu, R. G. (2006). Fault injection-based test case generation for SOA-oriented software. In *Proceedings of IEEE international conference on service operations and logistics, and informatics (SOLI)* (pp. 1070–1078). Shanghai, China.
16. Offutt, J., & Xu, W. (2004). Generating test cases for web services using data perturbation. In *Proceedings of workshop on testing, analysis and verification of web services (TAV-WEB)*, July 1–10.
17. Arenas, M., Fan, W., & Libkin, L. (2002). What's hard about XML schema constraints? In *Proceedings of international conference on database and expert systems applications (DEXA)* (pp. 269–278). Berlin, Heidelberg.
18. Voas, J., & McGraw, G. (1998). *Software fault injection: inoculating programs against errors*. New York: Wiley. ISBN 0-471-18381-4.
19. Kassab, L., & Voas, J. (1998). Agent trustworthiness. In *Proceedings of 4th workshop on mobile object systems: secure internet mobile computations* (pp. 121–133). Springer, July 20–24.

20. Farmer, W. M., Guttmann, J. D., & Swarup, V. (1996). Security for mobile agents: authentication and state appraisal. In *Proceedings of the fourth european symposium on research in computer security (ESORICS)* (pp. 118–130), September.
21. Borselius, N. (2002). Mobile agent security. *Electronics & Communication Engineering Journal, 14*(5), 211–218.
22. Vigna, G. (Ed.) (1999). *Mobile agents and security*. (Lecture Notes in Computer Science). Berlin: Springer-Verlag, Telos.
23. Aouadi, H., & Ahmed, M. B. (2005). Mobile agents security. In *Proceedings of IEEE 2nd international conference on mobile technology, applications and systems* (p. 6), November 15–17.
24. Claessens, J., Preneel, B., & Vandewalle, J. (2003). (How) can mobile agents do secure electronic transactions on untrusted hosts? a survey of the security issues and the current solutions. *ACM Transactions on Internet Technology (TOIT), 3*(1), 28–48.
25. Ge, H., & Tate, S. R. (2006). Efficient authenticated key-exchange for devices with a trusted manager. In *Proceedings of the third international conference on information technology: new generations (ITNG'06)* (pp. 198–203), April 10–12.
26. Young, A., & Yung, M. (1997). Sliding encryption: a cryptographic tool for mobile agents. In *Proceedings of the 4th international workshop on fast software encryption (FSE'97)* (pp. 230–241), January 20–22.
27. Kim, H., Baek, J., Lee, B., & Kim, K. (2001). Secret computation with secrets for mobile agent using one-time proxy signature. In *Proceedings of the 2001 symposium on cryptography and information security* (pp. 845–850).
28. Sun. *NetBeans*. Accessed on June 5, 2009, http://www.netbeans.org/.
29. Sun. *Java remote method invocation*, Accessed on June 5, 2009, http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp.
30. WSDL. *Web services description language*. Accessed on June 5, 2009, http://www.w3.org/TR/wsdl.
31. Biron, P. V., & Malhotra, A. (2001). *W3C recommendation XML schema part 2: datatypes*, Accessed on June 5, 2009, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502.
32. Zhang, J., & Chung, J.-Y. (2003). Mockup-driven fast-prototyping methodology for web application development. *Software Practice & Experience Journal, 33*(13), 1251–1272.
33. Sun. *J2EE*. Accessed on June 5, 2009, http://java.sun.com/javaee/.
34. Sun. *Sun java system application server*. Accessed on June 5, 2009, http://www.sun.com/software/products/appsrvr/previous/pe8/index.xml.
35. Sun. *Java web service development pack (JWSDP)*. Accessed on June 5, 2009, http://jwsdp.dev.java.net/index.html.
36. Sun. *JAX-RPC*. Accessed on June 5, 2009, http://jax-rpc.dev.java.net/.
37. Sun. *JAX-WS*. Accessed on June 5, 2009, http://jax-ws.dev.java.net/.

Author Biography

Jia Zhang received her Ph.D. degree in Computer Science from University of Illinois at Chicago in 2000. She is now an Associate Professor of Department of Computer Science at Northern Illinois University. Zhang has co-authored 1 book and has published over 100 refereed journal articles, book chapters, and conference papers. She is an Associate Editor of the IEEE Transactions on Services Computing (TSC) and International Journal of Web Services Research (JWSR). Zhang serves as Program Vice Chair of IEEE International Conference on Web Services (ICWS 2006–2009). Her current research interests center around Services Computing, with a focus on service oriented architecture, scientific collaboration, and services testing. She is a member of the IEEE.