Automated Testing and Response Analysis of Web Services

Evan Martin Suranjana Basu Tao Xie
North Carolina State University
Raleigh, NC, USA
{eemartin, sbasu2}@ncsu.edu, xie@csc.ncsu.edu

Abstract

Web services are a popular way of implementing a Service-Oriented Architecture (SOA), which has gained rapid adoption and support from leading companies in industry. Testing can be used to help assure both the correctness and robustness of a web service. Because manual testing is tedious, tools are needed to automate test generation and execution for web services. This paper presents a framework and its supporting tool for automatically generating and executing web-service requests and analyzing the subsequent request-response pairs. Given a service provider's Web Service Description Language (WSDL) specification, we first automatically generate necessary Java code to implement a client (service requestor). We then leverage automated unit test generation tools for Java to generate unit tests (including extreme, special, and random input values), and execute the generated unit tests, which in turn invoke the service under test. Finally we analyze the large number of request-response pairs from the web service invocation and identify robustness problems. We have applied our framework to freely available web services and our experiences show that we can quickly generate and execute web-service requests that may reveal robustness problems with no knowledge of the underlying web service implementation.

1. Introduction

Service-Oriented Architecture (SOA) [3, 12] is a software architectural style that aims to achieve loose coupling among interacting software agents. Service providers and service consumers are both implemented via software agents. A service is a unit of work done by a service provider to achieve some end result for a service consumer. Each service implements a specific business function and is made available such that the service can be accessed without knowledge of its underlying implementation. Furthermore, a single composite service may be implemented via several other services potentially owned and operated by different organizations. For example, a company may offer a service that allows its customers to search the product catalog. The company leverages the search service provided by Google to implement this functionality and thus relies on its correct operation. This scenario implies that the service provider becomes the service consumer. A service provider may not be willing to share implementation details, source code, or other intellectual property to facilitate web-service testing conducted by another company. As a result, the ability to perform black-box robustness testing is needed.

In this paper, we focus on testing web services, which are currently one of the most common technologies used to implement SOA. A web service is a self-contained software component with a well-defined interface that describes a set of operations that are accessible over the Internet. Web services can be implemented using any programming language on any platform, provided that a standardized XML interface description called Web Services Description Language (WSDL) is available *and* a standardized messaging protocol called Simple Object Access Protocol (SOAP) is used. Web services often run over HTTP but may run over other application layer transport protocols as well.

Important features of web services include SOAP, WSDL, and Universal Description, Discovery, and Integration (UDDI). These standard protocols are developed and used for web service selection and invocation at run time. WSDL is the XML format for describing network services. The UDDI registry information is used to locate a WSDL document, which can then be fed to a tool to generate a Java object proxy to the web service.

The challenge faced by a web service is ensuring its trustworthiness. The client-specific information to the web service is provided as input parameters. The instructions to manipulate the parameters are explicitly described within the WSDL document. However, if the web service is not robust enough to handle the parameters, malicious users can manipulate different parameter options in order to retrieve unauthorized information. Submitting extreme or random unexpected inputs can result in two scenarios: a web service can respond with “Illegal access exception” or it can allow illegal access to sensitive information in its database. In addition, if a web service responds with unexpected or uninformative exceptions, the client may crash while processing the response, causing robustness problems at the client side. Thus a web service’s robustness problem can pose a major challenge to web service security and reliability.

An example of this problem can be seen in an attack against Microsoft Excel identified by Guninski [17]. The imail 600 web service handles multithreading in such a way that if random malformed packets are used to flood the web ports, an attacker can crash the service and create large memory leaks. Such cases can cause serious concerns in high-risk operations like banking and defense. It is also important that a service delivers a function with the expected quality of service (QoS), which is important to build confidence among users and system integrators. Very often the integrators and the service providers agree to a Service Level Agreement (SLA) [10], which mandates the service provider to ensure the QoS. If a service does not meet the requirements of the mandated QoS, it can be considered in violation of the SLA. Heavy network or server load can be another reason for affecting the service performance. This factor further highlights the importance of ensuring robustness in service delivery. It is thus important to test the QoS of a web service, by generating various combinations of input and binding, as is done in web service testing.

However, testing web services poses several challenges. The unavailability of source code is a major challenge to white-box testing [7], as services are just provided as interfaces to the users and systems. This constraint makes black-box testing [7] the only viable solution. As a result, the users of the service very often have to rely on the service providers, without being able to easily ensure the robustness or reliability of the service themselves.

In this paper, we present a framework called WebSob and its supporting tool for automated robustness testing of web services. Given a description of the public interface to a service in WSDL, WebSob generates Java source code required for the service consumer to perform service requests on the service provider. WebSob also generates a wrapper class that maps a single method to each available service operation. This wrapper class is supplied to an existing test generation tool for Java programs such as JCrasher [11], which generates JUnit [16] tests. The execution of these unit tests automatically results in web-service requests to the service provider. WebSob then helps detect robustness problems by analyzing the responses from the web service. We have applied WebSob to 35 freely available web services and our experiences show that WebSob can quickly generate and execute web-service requests that reveal potential robustness problems in 15 web services with no knowledge of the underlying service implementation.

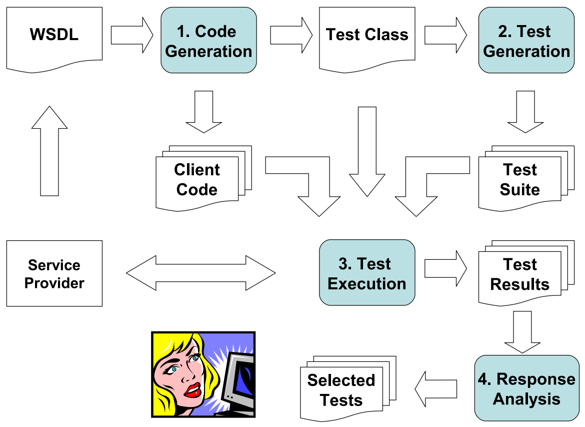

Figure 1. Overview of the framework

The remainder of the paper is organized as follows. Section 2 describes our WebSob framework and Section 3 presents implementation details of the framework. Section 4 shows our experience of applying WebSob on 35 freely available web services. Section 5 discusses potential issues and extensions to be addressed in future work. Section 6 presents related work and Section 7 concludes.

2. Framework

We have developed a framework called WebSob for robustness testing of web services, as illustrated in Figure 1. Given a WSDL from a service provider, WebSob first generates code to facilitate both test generation and test execution. WebSob then generates a test suite, which includes tests with random or extreme method-argument values. WebSob runs the generated test suite on the generated client code, which eventually invokes the web service. WebSob then collects the results returned from the web service. In particular, our WebSob framework consists of code generation, test generation, test execution, and response analysis. The code generation component generates necessary code required to implement a service consumer. In addition, the component generates a wrapper class that can execute each service independently. The wrapper class contains the service calls. JUnit tests are generated on this wrapper class to initiate SOAP requests to the service provider. The test generation component supplies the generated wrapper class to Java test generation tools in order to generate JUnit [16] tests. The test execution component executes the generated unit tests to cause the web service to be invoked and its responses to be collected. The response analysis component classifies and analyzes the collected responses from the web service. The component selects tests that cause the web service to return responses that are potentially robustness-problem-exposing.

2.1. Code Generation

The code generation component generates Java client-side code from a service provider's WSDL. The client-side code is used to allow unit-test generation tools to automatically generate unit tests that exercise the various services offered by the service provider as described in Section 2.2.

WSDL is an XML-based language that describes the public interface of a service. It defines the protocol bindings, message formats, and supported operations that are required to interact with the web services listed in its directory. In particular, the component generates the following classes or interfaces for each WSDL file: (1) A Java class is generated to encapsulate each supported message format for both the input and output parameters to and from the service. Message formats are typically described in terms of XML element information items and attribute information items. (2) A Java interface is generated to represent each port type, which defines the connection point to a web service. A stub class is generated for each binding, which defines the message format and protocol details for a web service. The binding element has the name and type attributes. The name defines the name of the service, while the port of the binding is defined by the type attribute. (3) A `Service` interface and corresponding implementation is generated for each service. The `Service` interface describes the exposed web services. This is a Java interface that facilitates web service invocation. (4) A wrapper class is generated to allow to invoke the provided service. This wrapper class contains the web service SOAP call to the operations supported by the web service.

2.2. Test Generation

The test generation component feeds the generated wrapper class to a Java unit-test generation tool to generate a test suite that exercises the services defined in the WSDL. The component operates relatively independently of test generation tools and thus any unit test generation tool for Java (such as JCrasher [11], Agitar Agitator [1], and Parasoft Jtest [23]) may be used. Because the wrapper class drives the test generation tool, it is important that the wrapper class is constructed such that it leverages the strengths of the chosen generation tool. For example, if information regarding pre-conditions, legal service invocation sequences, and/or functional specifications is known, then it is possible to encode this information in the wrapper class to further guide the test generation tool in producing better tests. Section 5 discusses examples of such cases in further detail.

Because our framework focuses on testing web services for robustness problems, this component intentionally adapts existing tools to generate extreme or special values besides random values for method arguments; these extreme or special values have a high probability of causing robustness problems. In particular, the component generates the following types of extreme or special values. Basic boundary values are explored for numeric data types such as -1, 0, and 1. For string-type arguments, the component generates strings of up to one hundred characters in length. In addition, in the generated strings, the component puts in some special characters such as "!@#". The component tries various possible combinations of several different special characters to form test inputs. For each non-primitive parameter, the component generates a null reference as a special argument value, which is eventually encoded by the SOAP engine via the xsi:nil attribute. In addition, the component constructs code fragments that will create instances of these complex data structures by passing different arguments to their Java constructors.

2.3. Test Execution

Given the generated wrapper class, unit-test suite, and client-side implementation, the test execution component simply runs the generated tests. These tests invoke methods in the wrapper class. The methods in the wrapper class have been designed to leverage the generated client-side implementation to invoke the remote web service under test. Since the web service is remote, and faults and exceptions are expected to occur, we set a timeout parameter in the wrapper class of one minute for the execution of each test in the generated test suite. This timeout mechanism ensures that the test driver does not hang indefinitely during execution.

2.4. Response Analysis

Manual inspection may be used to determine whether an exception should be considered to be caused by a bug in the web service implementation or the supplied inputs' violation of the service provider's preconditions. Even for the latter case, the web service implementation should respond with an informative error message rather than simply crashing information. For example, the GoogleSearchService has a string key argument that is used to uniquely identify the service requester. This identifier is received after registering with Google as a service consumer. If we allow this identifier to be automatically generated, then it is expected that the service should fail because the argument would violate the service provider's precondition, namely that the key identifier is valid. On the other hand, if the GoogleSearchService fails due to a generated search string, it is likely an issue since this interface should be prepared to handle such cases. The response analysis component selects tests whose responses may indicate robustness problems and presents the selected tests for manual inspection.

To collect web service responses, the component acts as a man-in-the-middle between the service consumer and the service provider. The service consumer directs the service request to the component, which records the request and forwards the request to the service provider. The component also records the service response or error condition returned by the service provider.

Based on our experience of applying WebSob on various web services, we classify four main types of encountered exceptions that may indicate robustness problems:

404 File Not Found. The 404 or Not Found error message is an HTTP standard response code indicating that the client was able to communicate with the server, but the server either could not find what was requested, or it was configured not to fulfill the request and not to reveal the reason why.

405 Method Not Allowed. The HTTP protocol defines methods to indicate the action to be performed on the web server for the particular URL resource identified by the client. 405 errors can be traced to configuration of the web server and security governing access to the content of the site.

500 Internal Server Exception. In certain cases, the server fails to handle a specific type of generated random input and produces an Internal Server Exception with an error code of 500. This exception is the most common one and offers little insight into what the problem may be.

Hang. The web service hangs indefinitely or the server takes more than one minute to return a response.

3. Implementation

In the code generation component, we leveraged Axis [4], a Java implementation of the SOAP protocol, to generate client-side code from a service provider's WSDL. In particular, the Axis utility class, `WSDL2Java`, parses the WSDL and generates necessary WSDL files that facilitate the implementation of service consumers.

In the test generation component, we modified JCrasher [11], a third-party test generation tool that automatically generates JUnit [16] tests for a given Java class. For example, JCrasher generates -1, 0, and 1 for arguments with the integer type and it can generate method sequences that create values for those arguments with non-primitive types. We have modified JCrasher to generate additional values for numeric arguments such as the maximum and minimum values supported by that type. JCrasher is designed as a robustness testing tool by causing the program under test to throw an undeclared runtime exception. More specifically, JCrasher examines the type information of a set of Java classes and constructs code fragments that create instances of different types to test the behavior of public methods. These code fragments are used in the generated unit tests to supply inputs to the public methods under test. In our case, the public methods under test are in the wrapper class. Each method there corresponds to a service defined in the WSDL and each method argument corresponds to an input parameter for that service. JCrasher generates unit tests that instantiate necessary input parameters to invoke the web service.

In the test execution component, we use JUnit [16] to execute the unit tests against the wrapper class, which invokes the remote web service. JUnit [16] is a regression testing framework that is used to execute a unit-test suite against the class under test.

In the response analysis component, we use TCPMonitor [5], which is the Axis utility provided to monitor the SOAP service being used. We pass the SOAP message information to the SOAP monitor service, which communicates with the target service to be tested. The SOAP message text is displayed through a web browser interface, by using an applet that opens a socket connection to the SOAP monitor service. Each time the SOAP connection is made to the local port, the request appears in the "Request" panel, and the response from the server appears in the "Response" panel. TCPMonitor also keeps a log of all the requests/response pairs.

4. Evaluation

We evaluated our WebSob framework by applying its implementation on selected WSDL files collected from the web. The following sections illustrates the evaluation of our framework.

4.1. Research Questions

Our evaluation tries to answer the following questions:

- Can WebSob generate test cases with only the WSDL file provided as input?
- Can WebSob identify potential robustness problems in web services?

4.2. Subjects

We have applied WebSob on the 35 freely available web services listed in Table 1. The first column lists the location of the WSDL file and the last four columns indicate the four categories described in Section 2.4. Thousands of requests have been quickly generated and executed for each web service. The web services used in our evaluation have been collected from public web sources that are

Table 1. WSDL files used to detect robustness problems of web services.

WSDL	Input	404	405	500	hang
`http://api.google.com/GoogleSearch.wsdl`	SC			√	
`http://developers.sun.com/sw/building/tech_articles/jaxrpc/AirService.wsdl`					
`http://oneoutbox.com/wsdl/FaxService.wsdl`	NULL			√	
`http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl`					
`http://webservices.imacination.com/distance/Distance.jws?wsdl`					
`http://ws.strikeiron.com/InnerGears/ForecastByZip2?WSDL`					
`http://ws.strikeiron.com/InnerGears/MetarBreakDown2?WSDL`					
`http://ws.strikeiron.com/LMStockQuotes/LMSecurities?WSDL`					
`http://www.ebi.ac.uk/webservices/chebi/webservice?wsdl`	NULL			√	
`http://www.ebi.ac.uk/webservices/wsintegr8/integr8?wsdl`	NULL	√	√	√	
`http://www.ebi.ac.uk/xembl/XEMBL.wsdl`					
`http://www.oneoutbox.com/wsdl/FaxStatus.wsdl`					
`http://www.oneoutbox.com/wsdl/FreeFaxService.wsdl`	LS			√	
`http://www.ripedev.com/webservices/ZipCode.asmx?wsdl`	NULL			√	
`http://www.synapticdigital.com/webservice/public/regions.asmx?wsdl`	NULL			√	
`http://www.unifiedsoftware.co.uk/BankValinternationalall.wsdl`					
`http://www.webservicex.com/airport.asmx?wsdl`	NULL			√	
`http://www.webservicex.com/CurrencyConvertor.asmx?wsdl`					
`http://www.webservicex.com/geoipservice.asmx?WSDL`					
`http://www.webservicex.com/globalweather.asmx?wsdl`	NULL			√	
`http://www.webservicex.com/stockquote.asmx?WSDL`					
`http://www.webservicex.net/LondonGoldFix.asmx?WSDL`					
`http://www.xignite.com/xcompensation.asmx?wsdl`	NULL			√	
`http://www.xmethods.net/sd/2001/BabelFishService.wsdl`	NULL			√	√
`http://www.xmethods.net/sd/CurrencyExchangeService.wsdl`	NULL			√	√
`http://xml.nig.ac.jp/wsdl/ARSA.wsdl`					
`http://xml.nig.ac.jp/wsdl/Blast.wsdl`	NULL	√	√		
`http://xml.nig.ac.jp/wsdl/BlastDemo.wsdl`	NULL	√	√	√	
`http://xml.nig.ac.jp/wsdl/ClustalW.wsdl`					
`http://xml.nig.ac.jp/wsdl/DDBJ.wsdl`	NULL			√	
`http://xml.nig.ac.jp/wsdl/Ensembl.wsdl`					
`http://xml.nig.ac.jp/wsdl/Fasta.wsdl`	NULL	√	√	√	
`http://xml.nig.ac.jp/wsdl/GO.wsdl`					

freely available. There are some services the require authentication to access their server, such as the services provided by Google or Amazon, while many others do not require authentication. The WSDL files have some common identification tags that are used by our framework to generate tests. The common features include the services, which can be identified by the tag `wsdl:operation`, the inputs and the outputs to the service, which can be identified by `wsdl:input` and `wsdl:output`, and the sequence of messages that will be used for the operation, which can be identified by `wsdl:message`. The beginning of the WSDL files usually lists the elements that can be used for carrying out the service.

4.3. Process

As is discussed in Section 2, our framework generates the wrapper class containing calls to the services supported by the service provider. Our framework then generates test inputs. The generated inputs are then run and the test executions eventually invoke the services. JCrasher randomly selects from a set of inputs for each primitive data type. This set is easily modified if user-specific values are needed. Table 1 indicates special-character string input (SC), long string input (LS), and a null reference (NULL) as three categories of test inputs that have resulted in one of the four types of responses we are interested in. An SC input is identified as any string with non-alpha-numeric characters. An LS input is identified as any string of length greater than 100 characters. Null is encoded with the xsi:nil attribute.

4.4. Results

The last four columns of Table 1 shows four types of responses that indicate robustness problems. If a web service is exposed with one of the preceding four types of responses, we also list the type of exposing test inputs in Column 2. An error code of 500 is generated when an input is valid (whose corresponding request reaches the web service end) and the resultant output is displayed as the response. Very often a service fails for null-reference inputs. We observe that the major reason for an error code of 500 is when null references are passed as input. Very often the web services are not able to handle null references. It usually generates an exception stack trace. Sometimes the stack trace also provides insight to the database objects and accessed methods. Sometimes the web services respond with information that helps reveal likely reasons about the causes of errors. This case is observed in the Google search service (googleSearch.wsdl), which generates an error of invalid authentication key, when random inputs are provided as keys to the service. In many cases, without access to the service source code, it is difficult to determine whether the cause of the exception is a service-user error (i.e., the inputs violated some precondition) or a potential problem in the service implementation. However, in some cases, we can infer what the problem may be and what category it may fall under. For example, the Google search service requires a key issued by Google as an authentication mechanism. WebSob executed a set of tests for a static correct key and a set of tests where the key was generated by JCrasher. As expected, Google returned a 500 Internal Server Error when the key is incorrect with a meaningful message: `Exception from service object: Invalid authorization key.` Unfortunately few other web services provided meaningful error messages. A potential problem with the service implementation that could be vunerable to denial of service attacks may have been found in the `CurrencyExchangeService` hosted by `www.xmethods.net`. This web service returns the exchange rate between two currencies and hangs indefinitely for some inputs. In summary, our experience on applying WebSob on these 35 web services shows that WebSob is effective in generating tests that can expose cases leading to potential robustness problems.

4.5. Threats to Validity

The threats to external validity primarily include the degree to which the subject web services and adapted third-party test generation tools are representative of true practice. These threats could be reduced by more experiments on a greater number of web services and more third-party test generation tools in future work. The threats to internal validity are instrumentation effects that can bias our results. To reduce these effects, we apply our approach to each subject program and also manually inspect the generated wrapper code and test code. One threat to construct validity is that our experiment makes use of the manually classified response types to indicate robustness problems; some of these classified responses may not actually indicate serious robustness problems that may require actions from the service providers.

5. Discussion

We next discuss several aspects of the framework or its implementation that can be extended or improved in our future work.

5.1. Test Generation

Our current implementation uses random test generation enhanced with predefined extreme or special test inputs. In test execution history, some random or extreme inputs may cause robustness problems for some earlier tested web services. When generating test inputs for the current web service, we can try out those earlier robustness-problem-exposing test inputs of the same type first. Doing so can increase the chance of exposing robustness problems of the current web service given limited resource. This technique is especially useful when the web service under test poses limits on the number of times that it allows to process the requests from the same IP address within a certain period of time. The technique shares the same goal of ordering the fault-revealing tests to run first with test prioritization [20, 24]. However, our test prioritization operates on testing different web services whereas previous test prioritization approaches operate on testing different versions of the same program.

Recently semantics, such as preconditions and effects (also called postconditions), have been proposed to extend WSDL to form Web Service Semantics called WSDL-S [2]. We can extend the code generation component to transform the preconditions into Java runtime checking code. Then our test generation component can exploit test generation tools based on symbolic execution (such as JPF [19, 31], Symstra [33], and jCUTE [25, 26]) to generate test inputs to cover all feasible paths in the precondition runtime checking code, achieving full structural (precondition) specification coverage [9]. We can also manipulate the precondition runtime checking code to allow test inputs that violate the preconditions to be generated in order to test web services' robustness.

5.2. Other Types of Quality Attributes

Our current framework focuses on testing robustness of web services. At the same time, as we have discussed earlier in the paper, our framework can also help detect potential security vulnerabilities or performance issues to some extent. To enhance the framework's capability of detecting security vulnerabilities, we can incorporate domain knowledge of specific types of attacks in the test generation component. Different from previous attack generation approaches [27], the challenge here is that the test generation component has no access to the source code of the web service under test.

One important type of quality attributes is functional correctness. If effects (also called postconditions) are specified in WSDL-S [2], we can extend the code generation component to transform the effects into Java runtime checking code. Then the effect runtime checking code can check the responses from the web service under test against the effects. If there is any effect violation caused by responses for test inputs that satisfy the preconditions, a functional defect of the web service is detected.

When there is no effect specified for the web service, we can adapt some previous approaches for selecting the responses of the most suspicious test inputs for inspection. For example, the operational violation approach [34] was proposed to select automatically generated tests for inspection when specifications are not available. In particular, an initial test suite that exhibit normal service behavior (either a test suite released by the service provider, manually generated by the service consumer, or automatically generated by a test generation tool) is executed and the exercised request-response pairs are collected. Then we can exploit Daikon [13], a dynamic invariant detection, to infer operational abstractions (in the form of preconditions and effects) for the web service under test. Then we run the generated tests again the inferred operational abstractions. We can select any violating tests (tests that violates the operational abstractions) for inspection. These violating tests have a high probability of indicating service faults or failures.

5.3. Regression Testing

Regression testing of a web service tries to assure that a new version of the web service behaves in the same way as the old version. However, when the service provider updates its web service version, there is usually no way for the service provider to notify service consumers [8]. Since our framework collects the request-response pairs from test executions, we can conduct regression testing on the web service under test by running the same test inputs multiple times periodically and analyzing the request-response pairs to detect regression errors. But the responses of some web services may depend on other environmental factors (e.g., time, date, and other real-time data sources) besides the received requests. We can do statistical analysis on request-response pairs across versions (i.e., different times of running the same tests, because service consumers have no information on whether the web service version has been changed) to identify these types of environment-dependent web services. Then in regression testing, we compare responses of the same requests across versions except for these environment-dependent web services. An alternative way for doing regression behavior checking is to directly embed the response-checking assertions in the generated tests by turning previously captured responses into assertions in test code [32].

6. Related Work

Fu et al. [15] developed a tool for testing Java web services at the service-provider site. Their tool injects faults into service implementations to conduct white-box coverage testing of error recovery code such as exception handlers. Unlike their tool, our tool conducts testing at the service-consumer site, without the access to service implementations.

Other previous work on testing web services is primarily model-based testing. Nakajima [21] proposed a model checking technique that verifies service flows using a model checker. Narayanan and McIlraith [22] adopted the DAML-S ontology to describe web service semantics and test web services by simulating their execution under different input conditions. Foster et al. [14] proposed a finite-state process notation to model and verify service composition. Yi et al. [35] proposed an extended Petri Net model for specifying and verifying web service composition. Tsai et al. [18,29,30] proposed test-case generation based on OWL-S specification. Our approach to testing web services does not require models.

Bai et al. [6] proposed a WSDL-based method of test-case generation, execution, and response analysis. Sneed and Huang [28] developed the WSDLTest tool for generating random requests for WSDL schemas, adjusting the generated requests with manually written preconditions, dispatching the adjusted requests, collecting responses, and verifying the collected responses against manually written postconditions. While each of these approaches is similar to ours, to our knowledge, no other test generation tools for web services leverages existing Java test generation tools and the JUnit framework for test execution. Doing so allows our approach to exploit state-of-the-art Java test generation tools to conduct web service testing.

7. Conclusions

We have developed the WebSob framework and its supporting tool for automatically generating and invoking web services given a service provider's WSDL. WebSob first generates necessary code to implement a client (service requestor) along with a wrapper class. WebSob then leverages existing automated unit test generation tools to generate unit tests for the wrapper class and finally executes the generated unit tests, which in turn invoke the service under test. Then WebSob collects and analyzes the responses from the web services in order to detect robustness problems. Our experiences of applying WebSob on 35 web services showed that WebSob can quickly generate a large number of web-service tests and successfully execute them against a service provider. These requests reveal robustness problems of 15 web services with no knowledge of the underlying service implementation. Such a tool to web services testing is useful when a service provider also relies on a third-party service provider to function correctly.

References

[1] Agitar. Agitar Agitatior 2.0, Novermber 2004. `http://www.agitar.com/`.

[2] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M.-T. Schmidt, A. Sheth, and K. Verma. Web Service Semantics - WSDL-S Version 1.0. `http://www.w3.org/Submission/WSDL-S/`, November 2005.

[3] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services Concepts, Architectures and Applications Series: Data-Centric Systems and Applications*. Addison-Wesley Professional,, 2002.

[4] Apache. Axis. `http://ws.apache.org/axis/`.

[5] Apache. Axis TCP Monitor. `http://ws.apache.org/axis/java/user-guide.html#AppendixUsingTheAxisTCPMonitorTcpmon`.

[6] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen. WSDL-based automatic test case generation for web services testing. In *Proc. IEEE International Workshop on Service-Oriented System Engineering*, pages 215–220, 2005.

[7] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[8] G. Canfora and M. D. Penta. Testing services and service-centric systems: Challenges and opportunities. *IT Professional*, 8(2):10–17, 2006.

[9] J. Chang and D. J. Richardson. Structural specification-based testing: automated support and experimental evaluation. In *Proc. 7th ESEC/FSE*, pages 285–302, 1999.

[10] I. Corporation. WSLA Language Specification Version 1.0. `http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf`, January 2003.

[11] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.

[12] T. Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, 2005.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[14] H. Foster, J. Kramer, J. Magee, and S. Uchitel. Model-based verification of web service compositions. In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 152–16, 2003.

[15] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott. Testing of Java web services for robustness. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 23–34, 2004.

[16] E. Gamma and K. Beck. JUnit, 2003. `http://www.junit.org`.

[17] G. Guninski. `http://www.guninski.com/`.

[18] H. Huang, W.-T. Tsai, R. A. Paul, and Y. Chen. Automated model checking and testing for composite web services. In *Proc. 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 300–307, May 2005.

[19] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, April 2003.

[20] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proc. the 24th international conference on Software engineering*, pages 119–129, 2002.

[21] S. Nakajima. Model-checking verification for reliable web service. In *Proc. OOPSLA 2002 Workshop on Object-Oriented Web Services*, 2002.

[22] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *Proc. 11th International Conference on World Wide Web*, pages 77–88, 2002.

[23] Parasoft. Jtest. `http://www.parasoft.com`.

[24] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, 2001.

[25] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proc. 18th International Conference on Computer Aided Verification, Tool Paper*, pages 419–423, 2006.

[26] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 263–272, 2005.

[27] Y. Shin, L. Williams, and T. Xie. SQLUnitGen: SQL Injection Testing Using Static and Dynamic Analysis. In *Supplemental Proc. 17th IEEE International Conference on Software Reliability Engineering*, November 2006.

[28] H. M. Sneed and S. Huang. WSDLTest – a tool for testing web services. In *Proc. 8th IEEE Symposium on Web Site Evolution*, pages 14–21, 2006.

[29] W. T. Tsai, Y. Chen, and R. Paul. Specification-based verification and validation of web services and service-oriented operating systems. In *Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 139–147, 2005.

[30] W. T. Tsai, R. Paul, Y. Wang, C. Fan, and D. Wang. Extending WSDL to facilitate web services testing. In *Proc. 7th IEEE International Symposium on High Assurance Systems Engineering*, pages 171–172, 2002.

[31] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering*, pages 3–12, 2000.

[32] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *Proc. 20th European Conference on Object-Oriented Programming*, pages 380–403, July 2006.

[33] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, April 2005.

[34] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engineering Journal*, 13(3):345–371, July 2006.

[35] X. Yi and K. J. Kochut. A CP-nets-based design and verification framework for web services composition. In *Proc. IEEE International Conference on Web Services*, pages 756–766, 2004.