Automated Robustness Testing of Web Services

Evan Martin, Suranjana Basu, and Tao Xie

Department of Computer Science
North Carolina State University
Raleigh, NC 27695
{eemartin,sbasu2}@ncsu.edu,
xie@csc.ncsu.edu
http://ase.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 industrial players such as IBM, Oracle, and Microsoft. 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 new framework for automatically generating and executing web-service requests. Given a service provider's WSDL, we first generate the necessary code to implement a client (service requestor). We then leverage existing automated unit test generation tools to generate unit tests and finally execute the generated unit tests, which in turn invoke the service under test. Our preliminary results 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.

Key words: Web Services, Automated Testing, Service-Oriented Architecture

1 Introduction

Service-Oriented Architecture (SOA) 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 the most common technology used to implement SOA nowadays. 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.

In our research, we develop a framework for automated robustness testing of web services and its supporting tool. Given a description of the public interface to a service in WSDL, our tool generates code required for the service consumer to perform service requests on the service provider. The tool also generates a test class that maps a single method to each available service operation. This test class is supplied to an existing test generation tool for object-oriented programs such as JCrasher [4], which generates JUnit [7] tests. The execution of these unit tests automatically result in web-service requests to the service provider. Our preliminary results indicate that we can easily and automatically generate test suites for web services, designed specifically for robustness testing given only a service provider's WSDL.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 describes our approach to robustness testing of web services followed by some preliminary results in Section 4. Finally, Section 5 concludes with future work.

2 Related Work

Fu et al. [6] developed an approach for testing Java web services at the service-provider site. Their approach injects faults into service implementations to conduct white-box coverage testing of error recovery code such as exception handlers. Unlike their approach, our approach 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 [8] proposed a model checking technique that verifies service flows using a model checker. Narayanan et al. [9] adopted DAML-S ontology to describe web service semantics and test web services by simulating their execution under different input conditions. Foster et al. [5] proposed a finite-state process notation to model and verify service composition. Yi et al. [12] proposed an extended Petri Net model for specifying and verifying web service composition. Tsai et al. [11] proposed test-case generation based on OWL-S specification. Bai et al. [3] proposed a WSDL-based method of test-case generation, execution, and response analysis. But to our knowledge, no other test generation approach for web services leverages existing unit test generation tools and the JUnit [7] framework for test execution.

3 Framework

We develop a framework for robustness testing of web services, as is illustrated in Figure 1. Given a WSDL from a service provider, we first generate code to facilitate both test generation and test execution. In conjunction with the generated client code, a test suite is generated to invoke the web service and collect the results. In particular, our framework consists of the following three steps:

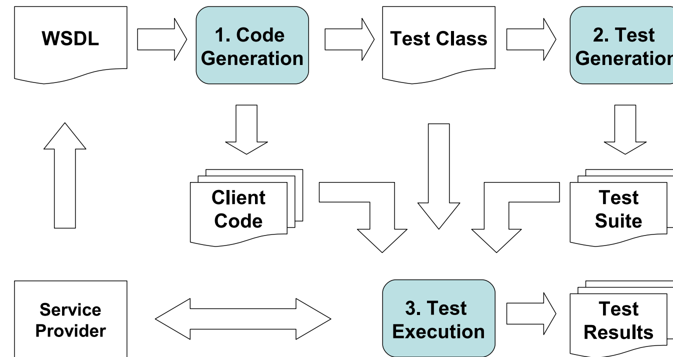

Fig. 1. Overview of the Framework

1. *Code Generation.* We generate the necessary code required to implement a service consumer. In addition, we generate a test class that can execute each service independently.
2. *Test Generation.* The generated test class is supplied to a test generation tool such as JCrasher [4] in order to generate JUnit [7] tests.
3. *Test Execution.* The execution of the generated unit tests causes the web service to be invoked and its responses to be collected.

3.1 Code Generation

Axis [2] provides a Java implementation of the SOAP protocol. We use Axis to generate client-side code from a service provider's WSDL. 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. The Axis utility class, `WSDL2Java`, parses the WSDL and generates the necessary WSDL files that facilitate the implementation of service consumers. A Java class is generated to encapsulate each supported message format for both the input and output parameters to and from the service. A Java interface is generated to represent each port type. A stub class is generated for each binding. A service interface and corresponding implementation is generated for each service. For our preliminary results, we have manually coded a wrapper class that leverages the code generated by `WSDL2Java`. Our future work plans to automate this manual effort by augmenting `WSDL2Java` to generate the required wrapper class. This wrapper class is designed 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 3.2.

3.2 Test Generation

Given the generated wrapper class, we use a unit-test generation tool to generate a test suite that exercises the services defined in the WSDL. Our approach operates independently of the test generation tool and thus other unit test generation tools (such as Agitar

Agitator [1] and Parasoft Jtest [10]) may also be used. Our preliminary results are obtained via JCrasher [4], a third-party test generation tool that automatically generates JUnit [7] 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. 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 the necessary input parameters to invoke the web service.

3.3 Test Execution

Given the generated wrapper class, unit test suite, and client-side implementation, we use JUnit [7] to execute the unit tests against the wrapper class, which invokes the remote web service. JUnit [7] is a regression testing framework that is used to execute a unit-test suite against the class under test. The test class throws an exception if a SOAP failure is encountered. Manual inspection and heuristics may be used to determine whether the 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.

4 Preliminary Results

We have successfully implemented our framework and applied it on web services provided by Google[1] and Amazon[2]. Thousands of requests have been quickly generated and executed; however, few interesting robustness problems have been detected. But we still observed that the Google search service occasionally appeared to hang indefinitely (though it is not clear why). Such anomalous behavior is a potential candidate for a bug report. The detection of a small number of potential robustness problems is likely due to the maturity and popularity of these services. We plan to further evaluate the approach on less mature web services in order to further validate its effectiveness for robustness testing.

5 Conclusions and Future Work

We have developed a framework for automatically generating and invoking web services given a service provider's WSDL. We first generate the necessary code to imple-

[1] `http://www.google.com/apis/index.html`
[2] `https://aws-portal.amazon.com/gp/aws/developer/registration/index.html`

ment a client (service requestor) along with a wrapper class. We then leverage existing automated unit test generation tools to generate unit tests for the wrapper class and finally execute the generated unit test cases, which in turn invoke the service under test. Our preliminary results show that we can quickly generate a large number of web-service tests and successfully execute them against a service provider. These requests may reveal robustness problems in the service provider's code with no knowledge of the underlying service implementation. Such an approach to web services testing is useful when a service provider also relies on a third-party service provider to function correctly.

In future work, we plan to evaluate our approach on several service providers, in hopes of revealing robustness problems in their implementations. We plan to automate the wrapper-class generation as well as response analysis. We also plan to incorporate more advanced unit test generation tools besides JCrasher [4]. Finally, we plan to provide tool supports for service providers in conducting white-box testing of service implementation.

With the increasing proliferation of web services, tools that facilitate their testing are increasingly important. We hope our work could help lead a trend in leveraging existing powerful tools for object-oriented programs to assist web service development, test, and deployment.

References

1. Agitar. Agitar Agitatior 2.0, Novermber 2004. `http://www.agitar.com/`.
2. Apache. Axis. `http://ws.apache.org/axis/`.
3. 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.
4. C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.
5. 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.
6. 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.
7. E. Gamma and K. Beck. JUnit, 2003. `http://www.junit.org`.
8. S. Nakajima. Model-checking verification for reliable web service. In *Proc. OOPSLA 2002 Workshop on Object-Oriented Web Services*, 2002.
9. 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.
10. Parasoft. Jtest. `http://www.parasoft.com`.
11. 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.
12. 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*, page 756, 2004.