Date of acceptance Grade

Instructor

Testing and Verification of RESTful Web Services

Ege Can Özer

Helsinki October 16, 2017

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Ege Can Özer

Työn nimi — Arbetets titel — Title
Testing and Verification of RESTful Web Services

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	October 16, 2017	11 pages + 11 appendices

Tiivistelmä — Referat — Abstract

Today, service-oriented architectures (SOA) are widely used and have become a major discipline for enterprise applications. Until the last decade, the most popular way to implement the services was using Simple Object Access Protocol (SOAP). Including the big companies such as Google, Facebook, Twitter, the direction moved towards to Representational State Transfer (REST) services due to the advantages such as its lightweight and scalability.

Unlike the conventional software testing, web services require different testing methods due to their loosely coupled, headless, and distributed architectures. In the literature, general trends and challenges of SOA testing reviewed, but the discussion primarily focused on the SOAP web services. Having said that there is a demand to demonstrate recent approaches concerning testing RESTful services.

This paper presents different means for testing and verification of RESTful web services, showing the advantages and disadvantages of testing tools and current approaches; and includes an analysis of five of this specialized methods from the service testing point of view. Based on the comparative results, we will identify issues for the future work.

ACM Computing Classification System (CCS):
Applied computing → Enterprise computing → Service-oriented architectures

Avainsanat — Nyckelord — Keywords
Service-oriented architectures, Software testing, Web services, REST

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Today, service-oriented architectures (SOA) are widely used and have become a major discipline for enterprise applications. Until the last decade, the most popular way to implement the services was using Simple Object Access Protocol (SOAP). Including the big companies such as Google, Facebook, Twitter, the direction moved towards to Representational State Transfer (REST) services due to the advantages such as its lightweight and scalability. In 2012, ProgrammableWeb reported that 75% out of all APIs follows REST architectural style, and it continues to grow exponentially [1].

Testing plays a critical role to ensure certain reliability and quality for SOAs. Unlike the conventional system-level testing, testing methods differ in service-centric systems. In the literature there are many articles presents several approaches to address the problems in SOA testing. Canfora and Di Penta [2] report a survey of SOA testing, they analyze the challenges from different stakeholders point of view and categorize them based on testing levels. Whereas, Bozkur et al. [3] extends the research by surveying 177 papers, identifies the features of testing strategies. However, in both of the surveys, the discussion primarily focuses on SOAP services.

Nevertheless, many of the testing related issues in SOA based web services are common and inherited by RESTful web services. Testing challenges in web services emerge from distributedness, loose-coupling, and lack of reliability of WWW as a common communication framework [4]. Moreover, headless (lack of graphical user interface) structure of the web services makes manual testing difficult to interpret. Many other challenges do also exist due to the complexity and the limitations that are imposed by the SOA environment [5, 2, 3]. Still, various strategies have been put forward to handle testing and validation of SOA, ranging from testing frameworks, model-based testing to evolutionary algorithms.

In this article, we present and give a brief analysis of five different testing approaches developed for RESTful services that are advanced during the last decade. First, Chakrabarti and Kumar [4] introduce a testing framework that contributes to many novel techniques. Second, Chakrabarti and Rodriquez [6] demonstrate a particular feature of RESTful web service testing, named *connectedness*, which is an essential aspect for resource-oriented architectures to assure. In the third article Pinheiro et al. [7] present a model-based testing approach using UML protocol state machine to generate test cases automatically. Next article by Navas et al. [8] show an automatic

error detection system based on statistical information without fully knowing its full-service specification. With that, they open an opportunity to apply Machine Learning algorithms to this area. The final article by Andrea Arcuri [9] propose an automated white-box testing approach, where test cases are generated using an evolutionary algorithm.

The rest of this document is organized as follows. In section 2, we survey the available different research around testing RESTful web services. Hence, section 2 is structured in two blocks. The first block presents the overview of system description and principal features of these five methods. Then in the following block, we provide a comparative analysis. In section 3, we will propose ideas for future research in this area. The last section summarizes and concludes the paper.

2 Testing methods for REST services

As far as testing web services are concerned much research has gone into investigating challenges stand in this area [2, 3]. In particularly for RESTful web services, the main challenges emerge from distributedness, loose-coupling, and lack of reliability of WWW as a common communication framework [4].

Distributedness architecture of the services often minimizes the controllability and observability of the system, and it makes locating the failures difficult. Lack of transparency complicates and enforces an unsystematic manual test case generation by the service testers. So that, designed test cases are only error-prone to developer's opinion [8]. Likewise, headless (lack of GUI) architecture of the web services poses the similar issue. Differently, loose-coupling makes the test case creation demanding because of data incompatibility between subsystems, especially if the working mechanism of the system is unknown. Extending the testing coverage by including other applications such as 3rd party web services and databases makes it even more complicated to test RESTful web services.

During the last years, there has been much progress related to testing RESTful web services. For example concerning the development of testing frameworks, *Test-the-rest* [4] demonstrated novel techniques to test RESTful services. Then following more specific approaches has undergone [6], such as testing the *connectedness* of the RESTful services. Subsequently, many of the approaches benefited from model-driven engineering methodology, Navas et al. [8] as an instance. As a result of recent advancements in the machine learning algorithms, new approaches [8, 9] have

also been adapted to this field. Since it is not possible to cover every aspect of the RESTful web service testing, which in general categorized in [2, 3], we are only going to focus on a small set of articles. Hence, the following section provides an overview of system description of the articles as mentioned earlier.

2.1 System description and principal features

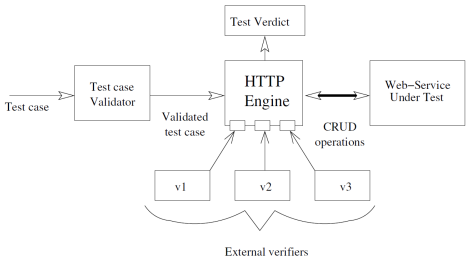
Test-the-rest: An approach to testing restful web services [4]. The first article authored by Chakrabarti and Kumar [4] presents a black-box testing framework called *Test-the-rest (TTR)*. The testing framework carries the following features: the capability to do functional and non-functional testing, automatic test case generation, extensible architecture design for testing RESTful web services, specialized test specification language that supports reusable and composable test cases.

TTR designed architecturally similar to WebInject [10] tool as the idea of writing test cases based on external XML files supports the automatic test generation. The general architecture of TTR is shown in Figure 1(a). According to this, first, the input test case has written go through a sanity check by the test case validator. Then HTTP engine issues the HTTP methods given under the test case. HTTP responses are validated for each test case by the external verifiers (e.g., XML validator plugin) and the test verdict is given.

However, the main contribution is the test specification language. The test specification language is XML formatted and similar to WebInject's test case language and Web Application Description Language (WADL). It supports composite test case validation by making use of logical operators, test case composition by allowing reuse and nested testing. Parametrization is also supported so that test cases can be generated with an arbitrary input automatically. An example of test case given in Figure 1(b).

Experiments took place in an internal infrastructure of authors'. In the first phase 300 manual test cases written by the testers, and tested daily where 5-10 test cases failed daily corresponding to real system bugs. In the second phase, test case generation is automatized and generated 42,767 test cases where the 38,016 test cases failed due to two significant bugs. After fixing the bugs, the number of failures reduced to 1781 identifiable bug.

Connectedness testing of RESTful web services [6]. The second article presents an algorithm to test connectedness of the RESTful web services [6]. The



(a) General architecture of TTR

```
<testcase>
  <id>TTR-COMPOSITE-VALIDATION</id>
  <URI>http://localhost/cgi-bin/hello-sujit.cgi</URI>
  <atomicity>atomic</atomicity>
  <method>GET</method>
<response>
  <representation mediaTypeExpression="xml | plaintext"/>
  <mediaType id="xml" value="XML"/>
  <mediaType id="plaintext" value="PLAINTEXT"
        src="D:\ResultValidationData.txt"/>
</response>
</testcase>
```

(b) Example test case

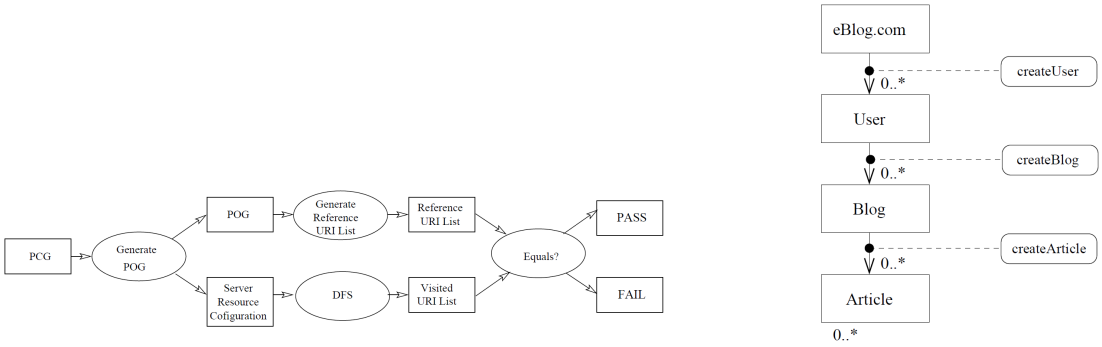
Figure 1: Figure 1(a) shows the general architecture of TTR. Figure 1(b) presents an example test case written in TTR's specification language. The representation field from the response can be either XML or plain text format. Having this flexibility is particularly useful when the media-type of the response representation is not known [4].

input of the algorithm requires a specification written in WADL++ that is an extension to WADL in defining a customized format for Post Class Graphs (PCG). PCG is a visual notation to describe the problem of connectedness, it is a directed graph whose nodes represent the resources, and the edges represent the HTTP POST operation. An example PCG shown in the Figure 2(b). WADL++ tailored for the connectedness problem in a way that it can describe which resource classes create which resource classes.

The main idea of testing connectedness is that after performing certain operations, there must be a one-to-one mapping between referenced URI list and visited URI list. If the condition meets the test passes, otherwise it fails. The overall procedure to test the connectedness of the RESTful web service shown in the Figure 2(a). So that the approach uses PCG to generate reference URI list, and it produces related server-side resource scenario to generate visited URL list. Finally, it checks if the two outputs are equal or not.

Experiments took place in a prototype of blogging web service called *eblog*. The approach generated 16 referenced URI for comparison, and it raised exceptions for all cases in the earlier development of *eblog*. Overall, the proposed tool helped in the early stage of development of *eblog* to identify errors regarding broken links.

Model-Based Testing of RESTful Web Services Using UML Protocol State Machines [7]. So far we have seen that previous articles make use of



(a) (b)

Figure 2: Overall procedure of connectedness testing shown in Figure 2(a), and Figure 2(b) gives an example of Post Class Graph that belongs to prototype web service [6].

specification-based testing. In this article Pinheiro et al. [7] presents a model-based method using UML protocol state machine to test RESTful web services.

The proposed system consists of many components, that is summarized by Figure 3. First, the tester creates the target system's behavioral model, that is protocol state machine. Then the model is converted to XML Metadata Interchange (XMI) format, which is a well-covered standard format among the modeling tools. After, each model is represented by Directed Acyclic Graph (DAG). DAG reflects only the information that belongs to system's certain states (initial, transition, and final). To generate test cases and related Java classes, the test case generation step requires test sequences and Abstract Syntactic Tree (AST) to be generated. AST is generated using expression parser from a given DAG. Furthermore, test sequences can be generated using two coverage algorithm (the tester decides which one to be used). The state coverage algorithm generates the test sequences based on DAG, whereas transition coverage algorithm based on AST. Finally, the system outputs 4 Java test classes to be able to test the system automatically.

The proposed system presents a convenient way to make model-based testing for RESTful web services. However, no experimental evaluations about the system have been carried out.

REST service testing based on inferred XML schemas [8]. In this article Navas et al. [8] present black-box automatic error detection system for RESTful web services based on statistical information. The method attempts to test web services without knowing its full specification. Also, with this approach authors have opened

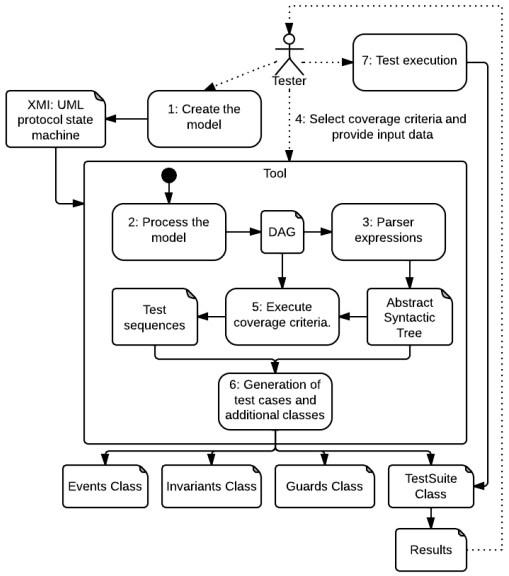


Figure 3: Model-based RESTful web service testing approach by Pinheiro et al. [7].

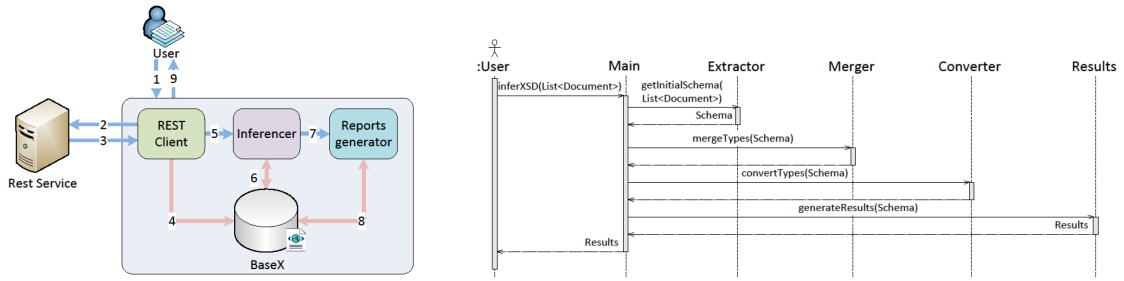
an opportunity for this area to employ Machine Learning algorithms.

The first procedure of the system starts with a user giving an input set of test data that consists of test URL, HTTP methods, and a set of parameters for the request. Then, the REST client module reads the data and generates related REST requests to the target web service. Each request-response pairs are stored in the database for later usage of inference module. Once the REST client tells the database location to inference module, the inferencer operates and collects the result. In the end, report generator module searches for any outliers that are obtained from the inferencer. Figure 4(a) presents the overall system architecture.

The inference module is responsible for building an XML Schema Definition (XSD) schema such that the all given documents are valid; additionally, during the process, it collects statistics about the document contents such as the number of appearances of unique elements and distribution of values for that element. It can be summarized accordingly by the given Figure 4(b).

Two types of experiments conducted to evaluate the system, one for the inference module, and other for the error detection system. The system is tested in Google places and configuration recovery service (their service). In the first experiment, they have compared their inference engine with existing tools Trand and SoapUI. In all cases, the inference engine surpassed the existing tools. In the second experiment no anomalies have found for Google places; however, 16 anomalies are detected for

the configuration recovery service with a 56.25% precision rate.



(a) (b)

Figure 4: Figure 4(a) shows the overall system architecture, and Figure 4(b) presents the sequence diagram of inferencer module [8].

RESTful API Automated Test Case Generation [9]. The final paper by Andrea Arcuri [9] presents a novel method to generate test cases for RESTful web services automatically. It focuses on white-box testing for an isolated environment. The novel method utilizes the evolutionary algorithm to generate test cases for the system under test (SUT). The implemented tool can tell found bugs in the SUT as well as coverage percentage.

The first step is the configuration of the SUT with the proposed tool to facilitate the test case generation process and the testing itself. The configuration is mainly related to information about the SUT such as location, packages to test, authentication methods, and so forth. It is also required to handle the instrumentation of the SUT to collect coverage data, extraction of methods available in the SUT, and to control the testing service remotely. Once the SUT is configured, the testing tool can generate test cases and test the SUT automatically.

The test case generation consists of three parts: definition, validation, and generation of the test cases. A test case can be defined automatically using a REST documentation tool called Swagger. Swagger can extract the available HTTP methods defined in the SUT, and outputs in JSON format. An example output of GET operation from Swagger is shown in the figure 5. HTTP response codes used to validate the status of a test case. So if the response belongs one of the 5XX server error group codes (e.g., 500 internal server error), it means that the test case failed. The final step of the method is the generation of the collection of test cases using the evolutionary algorithm, more precisely *Whole Test Suite* approach [11].

The implemented tool experimented on three different small to medium sized web

services of which two of them are open source, and the other is an industrial project. In total, the tool generated 176 test cases. 39 test cases, with one exception, led to identifying the bugs. The exception was due to the expected behavior of a web service that returned a 500 code. On the other hand, the obtained coverage (18% - 41% coverage) was significantly lower than the manual coverage result (41% - 82%). After the analysis of the underlined coverage problem, the author stated that improving string constraints, control over database operations and external services would result in a better coverage.

Figure 5: From left to right, extracted JSON definition of GET operation on */v1/activities/id*, its parameters, and the default response defined [9].

2.2 System analysis

In the literature, there are various efforts to describe and compare the different testing systems [5, 2, 3]. Since the main focus is on the different spectrum of testing approaches, neither of survey techniques in the previous articles are not directly feasible for our document. Besides, we believe that it does not give enough detail concerning practical implications. Given the fact, comparative analysis of the five RESTful web service testing methods is focused on mainly to following criterion: methodology, practicality, and evaluation.

Methodology comparisons and critics.

Practicality comparisons and critics.

Evaluation comparisons and critics.

Explanation of the table.

	Criteria	Chakrabarti and Kumar [4]	Chakrabarti and Rodriquez [6]	Pinheiro et al. [7]	Navas et al. [8]	Andrea Arcuri [9]
Methodology	Brief description	A framework to test RESTful web services	A method to test the connectedness of RESTful w.s.	A model-based approach to test RESTful w. s. using UML protocol state machine	An error detection system for REST based on statistical analysis	An automated test case generation using evolutionary algorithm for RESTful w. s.
	Testing method (box-approach)	Black-box	Black-box	Black-box	Black-box	White-box
	Testing level	Supports all testing levels	Non-functional testing *(connectedness)*	Functional testing	Functional testing	Integration, regression
	Test case generation	Yes	Yes	Yes	Tests are given as input	Yes
	3rd party tools used	None	None	JUnit, ArgoUML, JavaCC, JJTree, CodeModel, HttpComponents	BaseX XML database	Swagger, IntelliJ Coverage, EvoSuite
	Limitations	Immature compared to existing alternatives	Immature compared to existing alternatives	Complexity of the tool	Misleading error rates on simple test cases	Evolution can mutate in a particular way
	Main contribution	An extensible test specification format	A formal notation to describe connectedness	Inducing UML protocol state machine to REST	Schema inferencer module of the system	Automatic test case generation for REST
Practicality	Controlled environment	Required	Required	Partially required	Not required	Required
	Platforms/Languages	C#, .NET platform	WADL++	XMI, Java	XSD, WADL, Java	Java, Kotlin
	Existing alternatives	SpecFlow, SoapUI	Xenu, LinkTiger, Audisto	GraphWalker, SpecExplorer, NModel	Microsoft XSD Inferencer, SoapUI, Trang	DataGenerator
	CI/CD compatibility	Yes	Yes	Yes	Yes	Limited, due to code coverage
	Manual effort concerns	Writing the test cases under proposed framework	Writing the test cases under WADL++	Modelling the specific behavior of the system	-	SUT has to be configured manually once
Evaluation	Experimental validation	1 in-house web service	1 in-house prototype project	No	1 internal, 1 industrial web services	2 open-source, 1 industrial web services
	Repeatability	No	No	No	No	Yes
	Performance measures	Yes	No	-	No	Yes
	Comparison against other tools	No	No	-	SoapUI, Trang	No
	Results	Tool can detect bugs and generate exhaustive test cases (Inadequate quantitative data).	Tool genereated 16 referenced URI and raised exceptions for all.	-	Inferencer validated all given documents. Errors found with a 56.26% precision.	176 test cases generated, 38 bugs identified. But, yielded low coverage rate.

Table 1: The comparison matrix presents ...

3 Future research

Talk about machine learning, search-based algorithms, evolutionary algorithms.

Production ready approaches = NO

Validation of approaches in the experiments.

4 Conclusion

Present the topic again, summarize, and conclude with a future tendency under the consideration of testing the rest.

References

1 ProgrammableWeb, "Programmableweb is an information and news source about the web as a programmable platform," 2005.

2 G. Canfora and M. Di Penta, "Service-oriented architectures testing: A survey," *Software Engineering*, pp. 78–105, 2009.

3 M. Bozkurt, M. Harman, and Y. Hassoun, "Testing and verification in service-oriented architecture: a survey," *Software Testing, Verification and Reliability*, vol. 23, no. 4, pp. 261–313, 2013.

4 S. K. Chakrabarti and P. Kumar, "Test-the-rest: An approach to testing restful web-services," in *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD'09. Computation World:*, pp. 302–308, IEEE, 2009.

5 G. Canfora and M. Di Penta, "Testing services and service-centric systems: Challenges and opportunities," *IT Professional*, vol. 8, no. 2, pp. 10–17, 2006.

6 S. K. Chakrabarti and R. Rodriquez, "Connectedness testing of restful web-services," in *Proceedings of the 3rd India software engineering conference*, pp. 143–152, ACM, 2010.

7 P. V. P. Pinheiro, A. T. Endo, and A. Simao, "Model-based testing of restful web services using uml protocol state machines," in *Brazilian Workshop on Systematic and Automated Software Testing*, 2013.

8 A. Navas, P. Capelastegui, F. Huertas, P. Alonso-Rodriguez, and J. C. Dueñas, "Rest service testing based on inferred xml schemas," *Network Protocols and Algorithms*, vol. 6, no. 2, pp. 6–21, 2014.

9 A. Arcuri, "Restful api automated test case generation," in *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*, pp. 9–20, IEEE, 2017.

10 WebInject, "Webinject is a free tool for automated testing of web applications and web services," 2004.

11 G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

12 SmartBear, “The complete api test automation framework for soap, rest and more,” 2005.