Defining RESTful Web Services Test Cases from UML Models

Alexandre L. Correa
Federal University of Rio de Janeiro
33 PUBLICATIONS **188** CITATIONS
SEE PROFILE

Thiago Souza
Federal University of Rio de Janeiro
10 PUBLICATIONS **3** CITATIONS
SEE PROFILE

Eber Schmitz
Federal University of Rio de Janeiro
54 PUBLICATIONS **103** CITATIONS
SEE PROFILE

Antonio Alencar
Federal University of Rio de Janeiro
45 PUBLICATIONS **194** CITATIONS
SEE PROFILE

Defining RESTful Web Services Test Cases from UML Models

Alexandre Luis Correa
Departamento de Informática Aplicada
UNIRIO
Rio de Janeiro - Brazil
alexandre.correa@uniriotec.br

Thiago Silva-de-Souza
Escola de Ciência e Tecnologia
Universidade do Grande Rio (UNIGRANRIO)
Duque de Caxias - Brazil
thiagoein@gmail.com

Eber Assis Schmitz,
Antonio Juarez Alencar
Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro - Brazil
eber@nce.ufrj.br,
juarezalencar@dcc.ufrj.br

Abstract— **This paper presents an approach for RESTful Web Services test case generation. RESTful Web Services have features that are not fully covered by traditional software testing techniques. The proposed approach uses model transformation techniques to generate platform independent test cases from UML class models enriched with Object Constraint Language (OCL) constraints. These test cases are then transformed into platform specific test cases that can be used to verify the implementation of CRUD RESTful Web Services.**

Keywords-software testing; RESTful web services; UML; OCL.

I. INTRODUCTION

Recently, a new category of web services, called RESTful Web Services, has emerged as the predominant Web service design model. RESTful Web Services follow the REpresentational State Transfer (REST) architectural style and explicitly use HTTP methods in a way that follows the protocol as defined by RFC 2616 [7].

RESTful Web Services manipulate resources. A resource is any information item accessible through a Universal Resource Identifier (URI). Resources are described by data and metadata in a MIME (Multipurpose Internet Mail Extensions) compliant representation [7]. Resources are manipulated by transferring representations between clients and servers using the operations specified in the HTTP protocol: POST creates a new resource; GET retrieves the current state of a resource in some representation; PUT replaces the current state of a resource; DELETE removes a resource [18]. Therefore, RESTful Web Services bind each operation of a resource to a specific HTTP method.

In service-oriented architecture environments, it is imperative that each service correctly performs its functions. In most software development endeavors, testing is an important quality control technique. The main purpose of software testing is to detect software failures [14]. Functional testing is a type of testing where test cases are defined from the functional specification of the software under test. Therefore, the effectiveness of the test cases is directly related to quality of the specification.

In general, RESTful web services are described in a service contract using the Web Application Description Language (WADL) format [15]. Therefore, a WADL descriptor is a service contract, represented in XML, which describes all the operations allowed on resources, the URI templates and the supported representation formats [18]. A WADL document, however, does not specify any business constraints that should be satisfied by service operations, since it focus on the specification of technical aspects. In data-oriented services, such as CRUD (Create, Retrieve, Update and Delete) services, a WADL document does not specifies neither the invariants of the domain, nor the pre and post-conditions of each operation. Therefore, in order to allow the generation of more effective test case suites, services must be described by richer semantic specifications.

Resources handled by RESTful web services can be represented in several formats. Thus, RESTful web services testing should take into account not only the information used in each operation, but also the representation format of that information. For example, an operation that retrieves a *Company* resource can return the result in several possible formats, e.g., JSON, XHTML. Therefore, the data formats used in the inputs and outputs of each service operation is an additional variable that must be considered in the design of test cases for RESTful web services.

In RESTful web services there is a natural mapping between the HTTP methods (POST, GET, PUT and DELETE) and most CRUD business operations exposed by a service. A problem that arises when the implementation details of a CRUD data service are not available is related to the precedence relations between CRUD operations, e.g., a resource can only be retrieved if it has already been created. Therefore, the generation of test cases for CRUD data services should take such precedence relations into account.

This paper proposes a specification-based approach for CRUD RESTful Web Services test case generation. Test cases are derived from services specified using UML and OCL (Object Constraint Language [11]. This paper is structured in four sections. Section 2 discusses the related work. Section 3 presents the proposed approach and Section 4 draws some conclusions.

II. Related Work

Several approaches for web services testing have been proposed. In [3], [2] and [6], the most relevant approaches are described and compared. Most of the existing web services testing approaches aim at deriving testing cases from a service specification. These approaches can be classified into two categories: a) testing based on standard specification of web services and b) testing based on semantic specification of services. In the former category, test cases are generated by parsing WSDL [17] and XML Schema documents, whilst in the latter category, test cases are generated from richer semantic specifications produced using a more expressive language such as the Web Ontology Language (OWL) [2].

Two recent studies have proposed approaches to test case generation based on contracts specified in Web Service Semantics (WSDL-S) [16] and OCL. Both approaches use methods based on combinatorial analysis to reduce the number of generated test cases providing satisfactory coverage. The approach described in [9] uses the Pair-Wise Testing (PWT) method, whilst the Askaruinisa and Abirami's approach [1] uses the Orthogonal Array Testing (OAT) method. However, such approaches are not directly applicable to CRUD RESTful Web Services Testing since they can only generate test cases for web services that: i) do not change the system state, ii) handle simple data types and iii) are based on WSDL-S format, which are specific for WS-* services.

RESTful Web Services testing is still an emerging area. Chakrabarti and Rodriquez [4] and Van Gilst and Reza [12] are two of the few published work in this area. The former describes an algorithm to automatically test the connectivity of RESTful web services via their URI address. The latter proposes a framework for RESTful web services testing based on the generation of perturbation data on input parameters present in the operation specification. This work, however, uses proprietary XML formats for the representation of the input parameters, which can hinder their acceptance.

III. Proposed Approach

The proposed approach aims at producing test cases for CRUD RESTful Web Services that handle resources based on complex data types. It follows Model-Driven Development approaches and, in particular, the PIM (Platform Independent Model) - PSM (Platform Dependent Model) structure defined in the Model-Driven Architecture (MDA) specification [10]. Test cases are derived from two models: a PIM model that specifies the service independent from the implementation technology, and a PSM model that specifies the aspects related to its implementation as a RESTful Web Service.

Sections A, B, C and D describes these approach using a simple example of CRUD services that maintain instances of *Course* and *Student* entities in a school domain.

A. Platform-Independent Model

The first activity of the approach corresponds to the elaboration of a Platform Independent Model (PIM). The first element of the PIM of a service is the class model. This model is composed by three main categories of classes: service classes; domain classes and representation classes.

A service class defines the CRUD operations that manipulate instances of a domain class. Each service class is annotated with the `<<crud>>` stereotype, defining at least four basic operations on the target entity (createXXX, retrieveXXX, updateXXX, deleteXXX, where XXX is the name of the target entity). Each operation is associated with a specific stereotype, i.e., `<<create>>`, `<<retrieve>>`, `<<update>>`, `<<delete>>`. Each domain class represents a business entity whose instances are handled by the operations of a service class, being annotated with the `<<entity>>` stereotype.

Representation classes are used to define input and output data structures used in service operations, similar to the Data Transfer Object pattern (DTO) [5]. For example, the *createStudent* operation receives an instance of the *StudentRep* class containing the input information used by the implementation to create the instance of the *Student* entity to be added to the service memory as a result. A representation usually contains a structure similar to its corresponding entity class, i.e., it is described by almost the same attributes. Representation classes are annotated with the `<<representation>>` stereotype. Figure 1 illustrates an excerpt of the model produced for two CRUD services of the example domain.

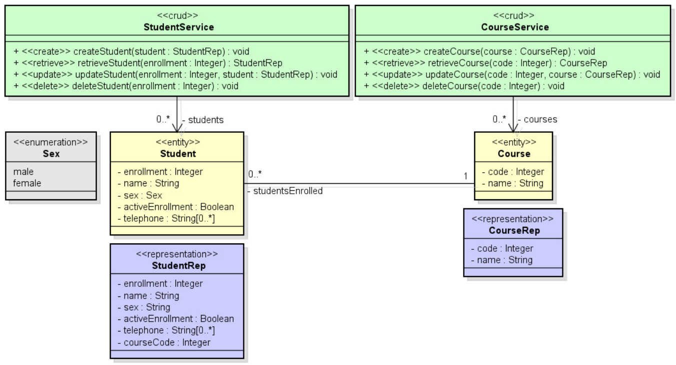

<<crud>>
StudentService
+ <<create>> createStudent(student : StudentRep) : void
+ <<retrieve>> retrieveStudent(enrollment : Integer) : StudentRep
+ <<update>> updateStudent(enrollment : Integer, student : StudentRep) : void
+ <<delete>> deleteStudent(enrollment : Integer) : void
<<crud>>
CourseService
+ <<create>> createCourse(course : CourseRep) : void
+ <<retrieve>> retrieveCourse(code : Integer) : CourseRep
+ <<update>> updateCourse(code : Integer, course : CourseRep) : void
+ <<delete>> deleteCourse(code : Integer) : void
0..*
- students
0..*
- courses
<<enumeration>>
Sex
male
female
<<entity>>
Student
- enrollment : Integer
- name : String
- sex : Sex
- activeEnrollment : Boolean
- telephone : String[0..*]
0..*
- studentsEnrolled
1
<<entity>>
Course
- code : Integer
- name : String
<<representation>>
StudentRep
- enrollment : Integer
- name : String
- sex : String
- activeEnrollment : Boolean
- telephone : String[0..*]
- courseCode : Integer
<<representation>>
CourseRep
- code : Integer
- name : String

Figure 1: Class model of School domain.

Rules constraining the values of the attributes of entity classes should be defined using invariants. These rules should be taken into account in the specification of the service operations. Figure 2 illustrates an example of this situation. In lines 1-2, an invariant defines the 1-99 range as the allowed values for the attribute *code* defined in the *Course* entity. The operation *createCourse* defined in the *CourseService* class, in turn, receives a data structure called *CourseRep*. As the invariant constrains the valid values for the *code* attribute, a pre-condition on the received value must be defined in the operation specification (lines 4-6).

```
1 context Course
2 inv: code >= 1 and code <= 99
3
4 context CourseService::createCourse(course :
5                          CourseRep)
6 pre: course.code >= 1 and course.code <= 99
```

Figure 2: Constraints on course code.

To avoid redundancy in the specification of constraints and preconditions of service operations, we suggest that constraints on attribute values should be specified in service classes as auxiliary operations. These operations can be referenced both in invariant and preconditions. Figure 3 shows an example of a refactored version of the constraints presented in Figure 2. The auxiliary operation *validCode* defined in the *CourseService* class (lines 1-3) is used in the specification of an invariant (lines 5-6) and a precondition (lines 8-10).

```
1 context CourseService::validCode(code : Integer) :
2                                  Boolean
3 body: code >= 1 and code <= 99
4
5 context Course
6 inv: courseService.validCode(code)
7
8 context CourseService::createCourse(course :
9                                     CourseRep)
10 pre: self.validCode(course.code)
```

Figure 3: Auxiliary operation used by an invariant and a pre-condition.

The PIM model should also specify the contract of each service operation using OCL pre and post-conditions. Since the operations of different CRUD services have similar behavior, e.g., create operations of *Student* and *Course* CRUD services are similar, we use templates, described in [13], to guide the writing process of contracts for each operation type. Figure 4 shows the contract for the *createCourse* operation, in which the preconditions (lines 3-5) state that a course can only be created if the arguments contain a valid code and a valid name (associated rules are defined as operations of the *CourseService* class) and if there is no course with the same code received as parameter (unique constraint on the course code). The post-condition (lines 8-14) ensures that, after the operation execution, a new instance will be added to the set of instances of *Course* existing at the invocation of that operation.

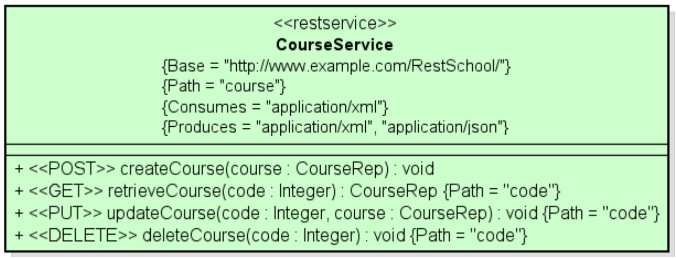
```
1 context CourseService::createCourse(cr :
2                                     CourseRep)
3   pre validCode: self.validCode(cr.code)
4   pre validName: self.validName(cr.name)
5   pre inexistentCourse: not courses->exists(c :
6                         Course| c.code = cr.code)
7
8   post newCourseCreated:
9    let newCourse : Course =
10     courses->select(c : Course | c.oclIsNew() and
11                    c.code = cr.code and
12                    c.name = cr.name)->asSequence()
13                                    ->first() in
14     courses = courses@pre->including(newCourse)
```

Figure 4: Contract of *createCourse*() operation.

B. Platform-Specific Model

The elaboration of the Platform Specific Model (PSM) consists in extending the PIM model by redefining stereotypes and adding tagged values to the service classes. The `<<crud>>` stereotype associated to classes in the PIM model is replaced by the `<<restservice>>` stereotype to indicate classes that implement RESTful web services. Stereotypes associated with operations in the PIM model are replaced by the `<<POST>>`, `<<GET>>`, `<<PUT>>` and `<<DELETE>>` stereotypes according to the operation type (create, retrieve, update, delete).

Tagged values may be used in two ways: (i) class scope and (ii) operation scope. In class scope, the tagged values "Base", "Path", "Consumes" and "Produces" are defined. "Base" indicates the base URI of the application, "Path" adds a relative path to base URI, "Consumes" and "Produces" respectively indicate the default representation formats consumed and produced by the service in all operations. In operation scope, the last three tagged values can be also used to define information specific to each operation. Figure 5 shows an example of stereotypes and tagged values defined in the *CourseService* class.



```
<<restservice>>
CourseService
{Base = "http://www.example.com/RestSchool/"}
{Path = "course"}
{Consumes = "application/xml"}
{Produces = "application/xml", "application/json"}

+ <<POST>> createCourse(course : CourseRep) : void
+ <<GET>> retrieveCourse(code : Integer) : CourseRep {Path = "code"}
+ <<PUT>> updateCourse(code : Integer, course : CourseRep) : void {Path = "code"}
+ <<DELETE>> deleteCourse(code : Integer) : void {Path = "code"}
```

Figure 5: Example of a class in the PSM model

C. Platform-Independent Test

The definition of PIT test cases is based on the constraints represented in the class model, e.g., attribute and association multiplicities, and on OCL constraints added to the PIM model. The technique is divided in two phases. In the first phase, a decision table derived from the constraints is created according to the following steps:

- Apply the boundary value analysis technique to the range of values defined by the invariants, enumerations and association multiplicities related to each entity class, separating them into valid and invalid values. Example for the *Course* entity: attribute code (null, 0, 1, 99, 100); attribute name (null, size 1, size 20, size 21).
- Generate entries in the decision table for valid input and invalid input combinations. Each row should correspond to the combination of at most one invalid value with valid values for the remaining input columns. This avoids combinations with more than one invalid value, since the expected behavior for one invalid input is the same as for more than one invalid input.
- Complete the decision table with the expected result for each entry. The expected result for combinations of valid values is success, whilst the expected result for rows containing an invalid input is error.

Table 1 shows the decision table generated to test the constraints on attribute values of the *Course* class.

Table 1: PIT decision table.

#	Input		Expected Result	
	code	name	Success	Error
1	null	size 1		X
2	0	size 1		X
3	1	null		X
4	1	size 1	X	
5	1	size 20	X	
6	1	size 21		X
7	99	size 1	X	
8	99	size 20	X	
9	100	size 1		X

In the second phase, test case procedures as defined as sequences of steps using the operations available in the interface of the service. Since test cases for CRUD services depend on the state of the service, defined by the existing instances and links between them, it is important to define a strategy to address the precedence relationships that exist in CRUD operations, i.e., in order to retrieve a course, we must ensure that this course has already been created. However, in black-box testing, we should rely only on the operations provided by the service.

One strategy used to minimize the number of steps per test case is the Chained Tests pattern described by Meszaros [8]. In this pattern tests are designed sequentially, so that the final state of a test is used as an initial state of the next test. However, the use of this pattern is not recommended since it violates the principle of independence between tests.

Our approach considers that each test case should create and clean up the state necessary for its execution. However, we only use the operations provided on the interface services, instead of using auxiliary test operations to control the state system, as suggested by *Back Door Manipulation* pattern [8]. Thus, one should consider that the service under test has an empty state and that each test case may use service operations to arrange the initial state (setup) and to clean up all state changes after test execution (tear down).

Table 2 shows a PIT test case for the *createCourse* operation whose steps are defined as calls to service operations. This example corresponds to the entry defined in line 4 of the decision table shown in Table 1.

Table 2: Example of PIT test case for *createCourse* operation.

Step Sequence	Expected Result	Step Function
1. retrieveCourse(1)	Error	Check pre-condition
2. createCourse({1,a})	Success	Execute main operation
3. retrieveCourse(1)	Success	Check post-condition
4. deleteCourse(1)	Success	Teardown

D. *Platform-Specific Test*

The PST test case generation translates the steps of the PIT test cases to HTTP requests, according to the appropriate method. Create operations are replaced by POST requests; read operations are transformed into GET requests; update operations are mapped into PUT requests; removal operations are represented by DELETE requests.

The verification of the expected results is performed by evaluating assertions on the response codes and headers of HTTP requests and responses. One or more assertion is done for each request, and the expected result is achieved when all assertions are true.

The technique considers in asserting the client response code for success and error (2xx and 4xx categories, respectively). Every HTTP response has a response code (status code), often used for error control. Table 3 shows the common response codes for success and error.

Table 3: Common response codes for HTTP verbs.

HTTP Verb	Success Code	Error Code
POST	201	400
GET	200	404
PUT	201	400
DELETE	200	400 or 404

The 201 status code used in POST and PUT methods indicates that the resource was successfully created or changed and that it can be referenced by a URI. The 200 status code, shared by GET and DELETE methods in successful HTTP responses indicates that the response contains the resource representation or that the resource was found and properly removed. The 400 status code used by POST, PUT and DELETE indicates that the operation was not performed because the client request contains a set of data that violates a rule operation. Finally, the 404 code used in GET and DELETE methods indicates that the resource was not found in the specified URI.

Assertions on error situations are based only in the status code. On the other hand, assertions of successful results to POST, GET and PUT methods also use some headers of the protocol itself. These headers are used to verify possible violations of technical constraints defined in the PSM.

POST requests use the Location header to indicate the URI of the resource just created. Thus, the verification of resource creation is not restricted to the status code. POST and PUT methods also uses the Content-Type header of the response to compare it with the Content-Type header of the request. This header indicates the representation formats supported by each operation (MIME types). So it is possible to check if the service implements correctly the configuration defined in the "Produces" and "Consumes" tagged values in PSM. These rules are shown in Table 4 which defines as a general definition of assertions associated to each HTTP method.

Table 4: General assertion model for PST test cases.

HTTP Verb	HTTP Success Response	HTTP Error Response
POST	*Status Code* = 201	*Status Code* = 400
	Location <> null	
	Content-Type = *Content-Type* of request	
GET	*Status Code* = 200	*Status Code* = 404
	Content-Type $\subset$ *Accept* of request	
PUT	*Status Code* = 201	*Status Code* = 400
	Content-Type = *Content-Type* of request	
DELETE	*Status Code* = 200	*Status Code* = 400 or 404

From this model of assertions it is possible to derive test case templates for each operation type. The operations defined in the PIT test cases should be transformed in order to use their corresponding HTTP methods and the corresponding assertions should be defined according to Table 4.

Table 5 shows the positive PST test case model for create operations. Line 1 performs a GET request, equivalent to a PIT retrieve operation, in order to ensure that the resource to be created does not exist in the service memory. Line 2 performs the main test case request, requesting the creation of a resource using valid data. Line 3 performs a new GET query to check if the resource has been created. The DELETE request in line 4 removes the resource created by the request of line 2, cleaning up the service state.

Table 5: Positive PST test case model for creating operations.

Request Sequence	Expected Result
1. GET	*Status Code* = 404
2. POST	*Status Code* = 201
	Location <> null
	Content-Type = *Content-Type* of request
3. GET	*Status Code* = 200
	Content-Type ⊂ *Accept* of request
4. DELETE	*Status Code* = 200

Based on the PIT defined in Table 2 and on the template shown in Table 5, Table 6 shows a positive test case implementation for *createCourse* operation. The URI is derived by concatenating the tagged values "Base" of *CourseService* class and "Path" of *createCourse* operation shown in Figure 5.

Table 6: Positive PST test case for *createCourse*() operation.

Request Sequence	Expected Result in the Response
GET http://www.example.com/RestSchool/course/1 Accept: application/xml, application/json	*Status Code* = 404
POST http://www.example.com/RestSchool/course Accept: application/xml, application/json Content-Type: application/xml <course> <code>1</code><name>a</name> </course>	*Status Code* = 201 *Location* <> null *Content-Type* = *Content-Type* of request
GET http://www.example.com/RestSchool/course/1 Accept: application/xml, application/json	*Status Code* = 200 *Content-Type* ⊂ *Accept* of request
DELETE http://www.example.com/RestSchool/course/1 Accept: application/xml, application/json	*Status Code* = 200

IV. Conclusion

This work presented a systematic approach for generating test cases for CRUD RESTful Web Services. The approach blends techniques focused on the specification of such services and techniques focused on the test case generation based on those specifications. The main contributions of this paper are related to the systematization of the production of test cases production for services implemented following the REST architecture.

Results collected by experimental studies showed that the quality of the test cases generated with the proposed approach is significantly better than those usually produced by students and test practitioners in industry. The proposed approach does not impose a specific tool for implementing the test cases, i.e., any tool or HTTP library can be used.

References

[1] A. Askaruinisa and A. M. Abirami. Test Case Reduction Technique for Semantic Based Web Services. International Journal on Computer Science and Engineering (IJCSE) , 02 (03), pp. 566-576, 2010.

[2] M. Bozkurt, M. Harman, and Y. Hassoun. Testing Web Services: A Survey. King's College London, Department of Computer Science, Londres, 2010.

[3] G. Canfora and M. Penta. Service-Oriented Architectures Testing: A Survey. Software Engineering: International Summer Schools, ISSSE 2006-2008 , pp. 78-105, 2009.

[4] S. K. Chakrabarti and R. Rodriquez. Connectedness testing of RESTful web-services. Proceedings of the 3rd India Software Engineering Conference (ISEC '10) (pp. 143-152). New York, NY: ACM, 2010.

[5] R. Daigneau. Service Design Patterns: fundamental design solutions for SOAP/WSDL and RESTful Web Services. Upper Saddle River: Pearson, 2012.

[6] A. T. Endo and A. D. Simão. Formal Testing Approaches for Service-Oriented Architectures and Web Services: A Systematic Review. Universidade de São Paulo, São Carlos, 2010.

[7] R. T. Fielding. Architectural styles and the design of network-based software architectures. PhD Thesis, Univ. of California, Irvine, 2000.

[8] G, Meszaros. xUnit Test Patterns: refactoring test code. Boston: Pearson, 2007.

[9] S. Noikajana and T. Suwannasart. An Improved Test Case Generation Method for Web Service Testing from WSDL-S and OCL with Pair-Wise Testing Technique. Proceedings of the 2009 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC '09) (pp. 115-123). Washington, DC: IEEE Computer Society, 2009.

[10] OMG, Object Management Group. MDA guide version 1.0.1. OMG, 2003.

[11] OMG, Object Management Group. Object Constraint Language, 2.2. 2010. http://www.omg.org/spec/OCL/2.2/PDF

[12] H. Reza and D. van Gilst. A Framework for Testing RESTful Web Services. Proceedings of the Seventh International Conference on Information Technology (pp. 216-221). IEEE Computer Society, 2010.

[13] T. Silva-de-Souza. Uma Abordagem Baseada em Especificação para Testes de Web Services RESTful. Master Thesis, PPGI, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2012.

[14] M. Utting and B Legeard. Practical Model-Based Testing: A Tools Approach. San Francisco: Morgan Kaufmann Publishers Inc, 2007.

[15] W3C, World Wide Web Consortium. Web Application Description Language. 2009. http://www.w3.org/Submission/wadl/.

[16] W3C, World Wide Web Consortium. Web Service Semantics - WSDL-S. 2005. http://www.w3.org/Submission/WSDL-S/.

[17] W3C, World Wide Web Consortium. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. 2007. http://www.w3.org/TR/wsdl20/.

[18] J. Webber, S. Parastatidis, and I. Robinson. REST in Practice. Sebastopol: O'Reilly, 2010.