Model-Based Testing of RESTful Web Services Using UML Protocol State Machines

Pedro Victor Pontes Pinheiro[1], **Andre Takeshi Endo**[1], **Adenilso Simao**[1]

[1]Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP) – 13.566-590 – São Carlos – SP – Brazil

Abstract. *Service-Oriented Architecture is a well-known architectural style that promotes many benefits among enterprise systems. In the last years, an alternative architectural style, so-called REST, has been proposed and widely adopted to design services' capabilities as resources. However, when it comes to verify these services, many challenges arise and hinder the process of testing. This paper proposes a model-based approach to test RESTful Web services using the UML protocol state machine as the formal behavioral model. A tool was developed to support the approach by automatically generating test cases for state and transition coverage criteria. An example is presented to illustrate the practical application of the approach.*

1. Introduction

Service-Oriented Architecture (SOA) is an architectural style employed by practitioners to develop scalable, flexible, and reusable enterprise systems. It has been recommended for supporting the development of complex and heterogeneous distributed systems [Josuttis, 2007]. SOA guides how a system should be structured so that its capabilities are exposed as black-box software units, known as services. In the last years, another architectural paradigm has been proposed and widely adopted to guide the structuring of a system in a more resource-oriented way, in which its capabilities are exposed not as services, but as resources. This architectural paradigm is called REpresentational State Transfer (REST), and systems that adopt its constraints are called RESTful Web services [Richardson and Ruby, 2007]

REST can be defined as an architectural style for hypermedia distributed systems like the World Wide Web (WWW) [Fielding, 2000]. It provides a set of constraints that, when applied to a system, minimizes latency and maximizes the components' scalability and independence. REST represents a solution to design and implement Web services and has been used as an alternative to the WS-* technologies, such as: Simple Object Access Protocol (SOAP) and Web Service Description Language (WSDL) [Richardson and Ruby, 2007]. Many IT companies have applied REST to implement their services, such as: Google[1], Yahoo[2], Facebook[3], Twitter[4], and LinkedIn[5]. Moreover, there is much interest in REST in several domains, such as: decentralized systems [Khare and Taylor, 2004], distributed simulation [Al-Zoubi and Wainer, 2009], cloud computing [Christensen, 2009], and relational databases [Marinos et al., 2010].

[1]`http://developers.google.com/custom-search/v1/usingrest`
[2]`http://developer.yahoo.com/search/rest.html`
[3]`http://developers.facebook.com/docs/reference/api`
[4]`http://dev.twitter.com/docs/api`
[5]`http://developer.linkedin.com/rest`

The testing process is fundamental to guarantee a certain level of reliability and quality of services based on REST. Testing RESTful Web services has its own challenges, making it harder to discover the faults. In general, Web services do not have a user interface, are loosely coupled and lack reliability on their communication framework [Chakrabarti and Kumar, 2009]. Other challenges also exist if the consumer is using a third party Web service, which is susceptible to functionality and availability changes without notification. Although there is a limited number of operations that can be executed over a resource (e.g., PUT, POST, DELETE, and GET), the states and transitions of resources in a RESTful service can be complex. In this context, UML protocol state machines have already been investigated to support the design of RESTful Web services [Porres and Rauf, 2011; Rauf and Porres, 2011; Rauf et al., 2010]. Although this kind of model can be used to automatically generate test cases [Xu et al., 2007], no initiative has been found in the application of UML protocol state machines to test RESTful services.

This paper describes a model-based testing (MBT) approach that facilitates the behavioral testing of RESTful Web services, respecting the REST constraints and providing a more systematic and formal testing. The chosen model to represent the system under test (SUT) was the UML protocol state machine [OMG, 2011] since it emphasizes the transitions between states, and not the actions that occur in each state, thus providing a level of abstraction that MBT needs [Xu et al., 2007]. Currently, test cases are generated based on two coverage criteria: state and transition coverage; the approach is illustrated through an example. Finally, a prototype tool that supports the approach automation is also described.The remainder of the paper is structured as follows. Section 2 provides the necessary background, introducing the concepts of RESTful Web services, and presents related studies found in the literature. Section 3 briefly introduces the UML protocol state machine and discusses its use to model RESTful Web services. Section 4 describes our approach to generate test cases for RESTful Web services out of UML protocol state machines. Section 5 brings information about the supporting tool. Finally, Section 6 draws the conclusion and sketches future work.

2. Background and Related Work

REST is an architectural style created by Fielding [2000] and consists of a set of design criteria, also called constraints. Services built on these criteria have a Resource-Oriented Architecture (ROA) [Richardson and Ruby, 2007]. The most important principles of REST are the following [Pautasso et al., 2008]:

- Resource identification through Uniform Resource Identifier (URI) – The resource (useful information to a client) must always have at least one URI associated with it. The URI can be both name and locator of the resource.
- Uniform interface – The client interacts with the resources using the four basic HTTP operations: POST, GET, PUT, and DELETE. Those operations correspond to create, retrieve, update, and delete, respectively. There are other operations like HEAD and OPTIONS that can be used. However, these operations only deal with the metadata about the resources.
- Self-descriptive messages – Resources can be accessed in a variety of formats, called resource representation, such as: XHTML, HTML, XML, JSON, plain text, and PDF. The server response, containing the resource, also present metadata that

can be used to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. The client can interact with the resource representations through different URIs.

- Stateless interactions – The interaction with the resources must be stateless (the statelessness property). That means that the server does not need to save data about the client. All the information necessary for the server to fulfill a request is included in the HTTP request. One resource can also point to other resources for future interactions.

Although the adoption of REST has been growing, few studies about testing of RESTful Web services can be found in the literature. We herein present the main studies related to our work. Chakrabarti and Kumar [2009] propose a test framework, called Test-the-REST (TTR), used to execute test cases based on specific REST requirements. The tester writes a test case which is represented as an XML file that has important pieces of information, such as: the HTTP method, the URI of the resource, and the expected representation. The results showed that the framework is capable of detecting many faults.

AlShahwan and Moessner [2010] conduct a study to compare the performance between SOAP-based frameworks and RESTful-based frameworks for mobile devices. The frameworks were evaluated in three different scenarios, and results show that RESTful-based frameworks are better suited for mobile environments. In the same context, Meng et al. [2009] conduct an in-depth performance analysis of traditional SOA-based Web services and RESTful Web services. Their testing scheme demonstrated that RESTful Web services are more suitable for Internet-scale distributed data integration. However, both works focus only on performance testing and do not apply to any type of functional testing, neither black-box nor white-box testing.

Reza and Van Gilst [2010] present a framework that provides test inputs to applications that integrate RESTful Web services. The framework basically provides a generalized test harness to simulate RESTful Web services that are not available to a developer, thus facilitating the process of unit testing of applications which rely on these Web services. The main steps of the framework consist of specifying the interface (available URIs, parameters and acceptable HTTP methods), specification of parameter values, and generation of test output.

Chakrabarti and Rodriquez [2010] present an algorithm to test whether a RESTful Web service conforms to the connectedness property of REST. Connectedness means that it is possible to access every other resource of a service from a root resource. To test that property, the algorithm uses two types of graphs that give important information about the URIs of the system: the POST Class Graph (PCG) and the POST Object Graph (POG). The PCG needs to be manually created, while the POG is automatically created by the algorithm. In this same context, Klein and Namjoshi [2011] formalize the concepts and properties of REST, which can be used to, e.g., model check the behavior of the SUT. Both of these works address the verification of REST constraints, but the former uses graph models, while the latter uses formal specification of the SUT.

Although SOA testing has been extensively investigated in literature [Bozkurt et al., 2012], most of the works focus on Web services based on WS-* technologies. In the context of RESTful Web services, we identified that existing research mainly focuses on test execution support, performance testing, and verification of REST constraints. There

is a lack of research that contributes for automated test case generation.

3. Modeling RESTful Web Services with Protocol State Machines

The UML state machine diagram (in UML 2) [OMG, 2011] is a type of diagram that models the different states of an object during the execution of a process. It is normally used to monitor the states in which an instance of a class goes through during its life cycle in response to events [Booch et al., 2005]. In UML, there are two types of state machines: behavioral state machines and protocol state machines. Behavioral state machines can be employed to model the behavior of individual entities, such as class instances. Protocol state machines are employed to express usage protocols and can be used to model life cycles of objects or the order of their operations [OMG, 2011].

Protocol state machines consist of states and transitions. The transitions are associated with a precondition (guard), postcondition and an event [Xu et al., 2007]. Protocol state machines are intended to model the sequence of events and the states caused by those events. In order to perform MBT, the chosen model must contain a certain level of precision and abstraction of the SUT [Utting et al., 2011]. The UML protocol state machine fits these criteria, because it provides a way to formalize, for example, the interface of classes, and they hide the behavior inside those classes. In other words, this model focuses more on the effect of the transition, rather than the behavior inside the states. It is important to highlight the fact that the UML notation contains object-oriented concepts that can conflict with the REST constraints [Schreier, 2011]. However, it is still possible to apply the model in RESTful services as shown in [Porres and Rauf, 2011] and [Rauf and Porres, 2011]. In these studies, modifications are made to the model to comply with the REST constraints. For example, the authors added state invariants using addressable resources to comply with the statelessness property and they also constrain the transition's events to the standard HTTP methods to comply with the uniform interface property.

A protocol state machine m is composed of different states S in which a system can be found. In the model, there is a set of transitions T that connect two states. So we can define the model as a tuple $m = (S, T, P, F)$ where P is a set of pseudostates and F is a set of final states. A transition $t \in T$ can be represented as a tuple $t = (s_i, e[p], s_j)$, where $s_i \in S$ is the state before the transition (source state), $e[p]$ is the event with the optional precondition p and $s_j \in S$ is the state after the transition (target state). Both source and target state can be the same ($s_i = s_j$). An event e can only be defined as a POST, PUT or DELETE operation since these operations are the only ones that can alter the state of the service. These operations must have a parameter that represents the resource, but POST and PUT operations can have additional parameters, in which their values are defined by the tester. There are two types of states: simple states and composite states. A simple state $s \in S$ can have an invariant inv; a composite state $s_c \in S$ can also have an invariant, but it is characterized by an associated state machine m' that is structurally within state s_c. An invariant inv is a Boolean expression on resources and attributes of its representation. There are two functions that can be invoked on a resource: OK and NOT_FOUND. The OK method returns true if the HTTP response status code obtained from a GET request is 200 (successful request); otherwise it returns false. The NOT_FOUND method returns true if the response status code is 404 (a resource was not found on the server); otherwise it returns false. Both methods accept the resource as parameter. An expression over a representation's attribute is similar to expressions used

in a precondition; this type of expression is explained later. A state is considered active if and only if its invariant evaluates to true. Invariants are also used as the postcondition of an event. According to the definitions herein presented, every protocol state machine is in fact a composite state, giving the possibility to apply an invariant to the whole model.

Figure 1 illustrates a protocol state machine model for a RESTful service that manages a hotel room booking (HRB). The service allows a client to book a room, pay for the reservation, and cancel it.

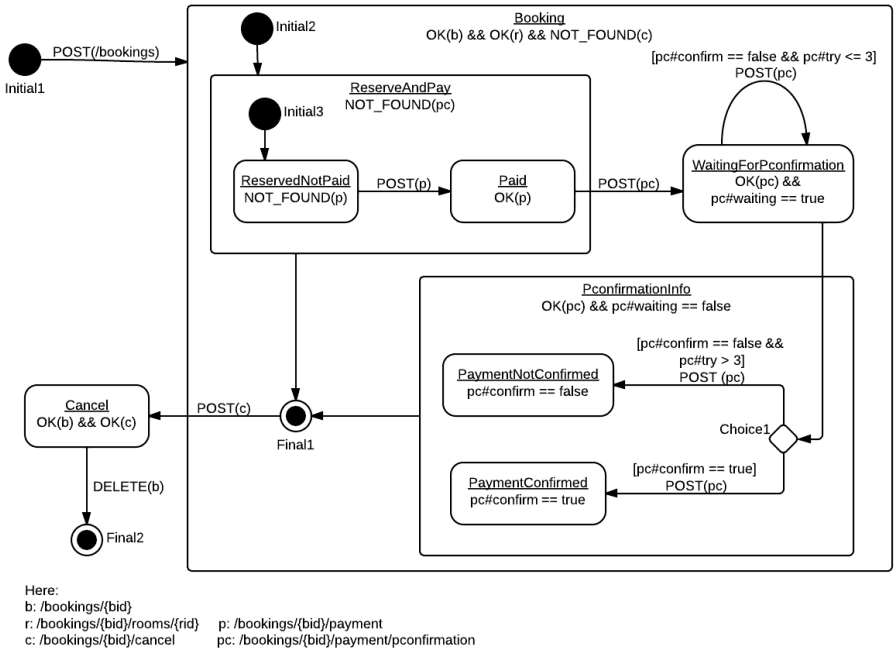

Figure 1. Behavioral Model of a RESTful Web service – adapted from [Porres and Rauf, 2011].

An invariant of a composite state must hold in its substates. For instance, the invariant "NOT_FOUND(pc)" of state *ReserveAndPay* is also applied to state *Paid*, resulting in the invariant "NOT_FOUND(pc) && OK(p)".

In the UML state machine, pseudostates are used to connect multiple transitions to form complex transition paths. Although there are 10 kinds of pseudostates [OMG, 2011], the ones used in our approach are only the initial and choice kinds. An initial state (Symbol $\bullet$) is used to indicate the default state for a composite state, and it cannot act as the target of a transition. A choice pseudostate (Symbol $\diamond$) represents a dynamic branching point. For instance, state *WaitingForPconfirmation* has a transition to a choice kind pseudostate, where the different preconditions must be analyzed to decide which transition should be enabled. A final state (Symbol $\odot$), in turn, is not considered a pseudostate, but a special kind of state which indicates finishing of its enclosing state [Liu et al., 2013].

A transition's precondition is expressed as a Boolean expression over resources' URI, resource's attributes, Boolean and relational operators, and constants. The resource attribute corresponds to a specific attribute in the representation. To access the attribute the format "resource#attribute" must be used. For instance, the expression "pc#confirm == true" in state *PaymentConfirmed* checks if the attribute "confirm" of the "pconfirmation" representation is true. This kind of expression can also be used as an invariant, as in

state *PconfirmationInfo*. Transitions that interact with a composite state have a specific behavior. If the source state of a transition is a composite state, it means that every substate is also a source state of that transition. If the target state of a transition is a composite state, then that state must have an initial state to indicate which substate will be the target of that transition.

4. Model-based Testing of RESTful Web Services

In this section, we propose a model-based approach to test RESTful Web services using the UML protocol state machine model described in the previous section. The testing approach we propose can be summarized by Figure 2.

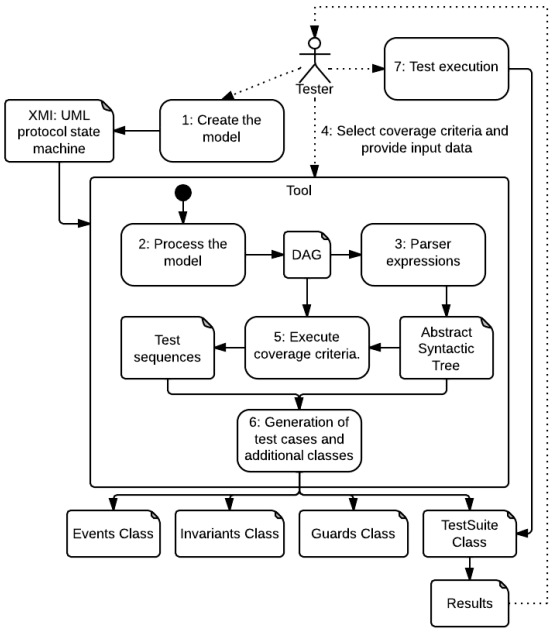

Figure 2. Model-based testing approach.

The starting point (Step 1) of the approach is to create the behavioral model (protocol state machine) of a specific behavior of the SUT. The behavioral model must then be converted to the XML Metadata Interchange (XMI) format. The XMI was chosen since it is a well adopted standard used in most modeling tools in the market. Usually the manual process of MBT demands too much effort from the tester, which is why tools are developed to assist most of the tasks. In our approach, a tool was developed to mainly read the information on the XMI and generate the test cases based on a specific coverage criterion. The test cases are generated in the Java language and use the JUnit[6] framework.

For each model there is a directed acyclic graph (DAG) that can represent it. A directed graph structure was chosen because it is a common structure to describe behavior in state machines and Petri Nets [OMG, 2011], and it is acyclic to prevent the execution of unnecessary paths. Figure 3 shows the DAG generated by the tool after the processing of the HRB model in Figure 1 (it corresponds to Step 2 in Figure 2).

This DAG only contains simple states, pseudostates and final states represented as nodes. For simplicity, some states are merged to form a single node, for instance, the node

[6] `http://junit.org`

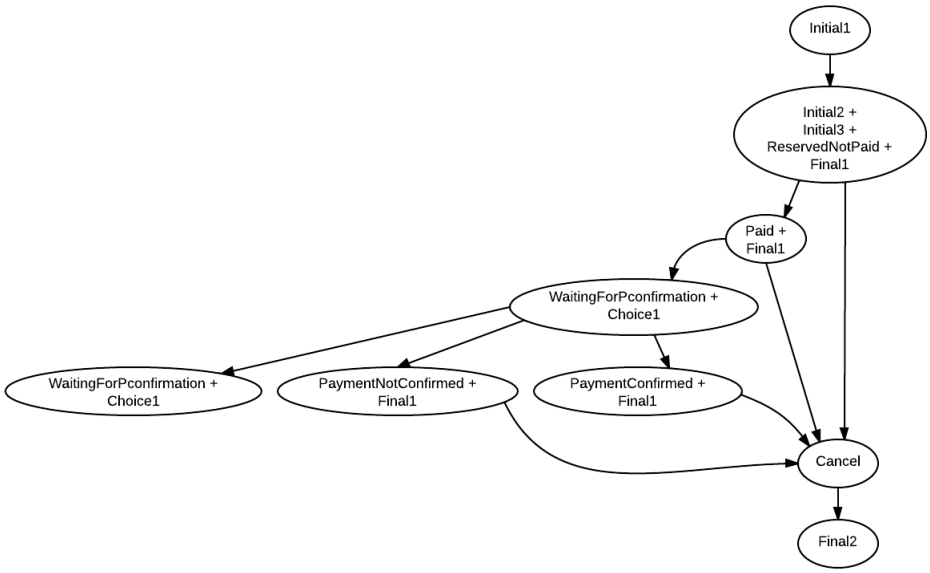

Figure 3. Corresponded DAG of the HRB model.

"Paid+Final1". Also, composite states are unfolded and their invariants and transitions are suited to the substates. Cycles are unfolded in such a way that each transition is traversed at least once. On Step 3, the tool uses a parser to evaluate all the expressions found in the model: invariant, and precondition expressions; and creates an abstract syntactic tree (AST). The tool, on Step 4, is also capable of detecting variables that must be provided by the tester (as input data). The tester also selects the coverage algorithm during that step. Variables can be used in the resource's URI, or in the event's parameters, for instance, the "bid" and "rid" variable.

Our tool supports two coverage criteria: the state coverage and the transition coverage. The algorithm for each criterion was based in the work of Xu et al. [2007] and Rauf and Porres [2011]. The state coverage algorithm generates the code based on the DAG as follows: *(i)* find the transitions that has as source the state represented by the current node; *(ii)* for each of these transitions traverse the node that its precondition can be satisfied, if the target state has not yet been traversed and if its invariant can be satisfied, then this state is marked as traversed; and *(iii)* expand the new node.

The transition coverage algorithm works as follows: *(i)* find the transitions that has as source the state represented by the current node; *(ii)* for each of these transitions traverse the node that its precondition can be satisfied, if the transition has not yet been traversed and if its invariant can be satisfied, then this transition is marked as traversed; and *(iii)* expand the new node. These algorithms are responsible to generate the test sequences (Step 5). A test sequence is a sequence of transitions $(s_0, e_0[p_0, q_0], s_1), (s_1, e_1[p_1, q_1], s_2), ..., (s_{n-1}, e_{n-1}[p_{n-1}, q_{n-1}], s_n)$, and it represents a test case.

Our supporting tool produces, as output, the required infrastructure to execute the test cases in the RESTful service under test (Step 6). The test sequences and the AST are used to generate four Java classes: Invariants Class, Events Class, Guards Class and TestSuite Class. Invariants Class has static Boolean methods that perform all the OK

and NOT_FOUND methods in the model. Events Class has static Boolean methods that perform all the POST, PUT and DELETE operations in the model. Each operation has its own HTTP status code to verify if it was performed successfully. A successful request returns a response code in the 2XX range. Guards Class has static Boolean methods that perform all the expressions that deal with representation's attributes. Finally, TestSuite Class contains the methods that execute the test sequences. Those classes are used by the main Java class, that contains the test cases, to perform events, check invariants and guards.

Then, the tester runs TestSuite Class on a command line or on an integrated development environment (Step 7). Our tool compiles the classes and organizes all the necessary libraries in a specific folder. Figure 4 shows a code snippet of a test case for a single transition. In the pseudocode it is possible to see that event "event1" only happens if precondition "guard1" evaluates to true; after that, postcondition "invariant1" is checked. Notice that any error situation in the test case is represented by some method returning false (events, guards, and invariants). In this case, the generated code is structured to fail the test case. If there is a failed test case, a fault can be identified and related to a specific part of the model.

```
@Test
public void testCaseExample() {
        if(Guards.guard1()) { // precondition
                assertEquals(true, Events.event1()); // event
                if(!Invariants.invariant1()) { //postcondition
                        fail();
                }
        } else {
                fail();
        }
}
```

Figure 4. Generated test case code example.

5. Supporting Tool

Our supporting tool was developed using the Java programming language. It has four main modules: state diagram construction, parsing, coverage algorithms, and code factory. During the phase of state diagram construction, the tool must read the XMI and be able to generate the directed graph. To do so, the tool uses the XPath framework to parser the XMI, and using the acquired information, creates the states and transitions. The composite states must then be unfolded. This operation must alter the transitions to point to the proper substates, and merge the invariants to substates as well. Finally, the tool runs an algorithm to convert directed graph to the proper DAG.

In the current stage of the prototype, we are using the ArgoUML[7] tool to create the behavioral model and generate its XMI; thus the tool is currently adapted to deal with the XMI generated by ArgoUML. The parser module was built using two frameworks: JavaCC and JJTree[8]. The module checks for any error in the expressions and builds the abstract syntactic tree, which is later used for code generation. The code factory works

[7] http://argouml.tigris.org
[8] http://javacc.java.net

along with the expressions to generate the classes and methods responsible for executing every expression. It also stores a relation between these expressions and the generated methods, which later can be used to generate the test cases. To generate proper Java code, the CodeModel[9] framework was also adopted. All the code interacts with the service using the HttpComponents[10] framework. Finally, the coverage algorithms are responsible to traverse the DAG and create the proper test cases using the information stored in the abstract syntactic tree.

6. Conclusion

We have presented a model-based testing approach to generate test cases for RESTful Web services. The approach uses an adapted UML protocol state machine that conforms to the resource-oriented architecture of REST. The current implementation of the tool generates test suites for state and transition coverage. We have presented an example to illustrate the practical application of the approach, from modeling to test execution. We believe that this initial effort motivates further research on model-based testing of RESTful Web services, from theoretical and experimental points of view.

In future work, the supporting tool can be extended to include automatic test data generation and other coverage criteria, as well as to deal with common features in most Web services like authentication, authorization, and resource representations (e.g., JSON, XML). Moreover, we intend to investigate how the approach can be applied to test the composition of RESTful Web services, taking into account existing advances in this topic [Rauf et al., 2010]. Oportunities to evaluate the approach are also manifold. It is important to analyze how the approach performs in the wild with real-world and industrial RESTful Web services. Finally, experimental evaluations (e.g. case studies) should be conducted in order to demonstrate the effectiveness of the proposed approach.

Acknowledgments

Andre T. Endo is financially supported by FAPESP/Brazil (grant 2012/21083-9). Adenilso Simao is partially funded by FAPESP (Grant 2012/02232-3) and CNPq (Grant 301170/2012-6).

References

Al-Zoubi, K. and Wainer, G. (2009). Using REST web-services architecture for distributed simulation. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS '09, pages 114–121.

AlShahwan, F. and Moessner, K. (2010). Providing SOAP web services and RESTful web services from mobile hosts. In *Internet and Web Applications and Services (ICIW), 2010 Fifth International Conference on*, pages 174–179.

Booch, G., Rumbaugh, J., and Jacobson, I. (2005). *The Unified Modeling Language User Guide*. Addison-Wesley, 2nd edition.

Bozkurt, M., Harman, M., and Hassoun, Y. (2012). Testing and verification in service-oriented architecture: a survey. *Software Testing, Verification and Reliability*, pages n/a–n/a.

[9] `http://codemodel.java.net`

[10] `http://hc.apache.org/httpcomponents-client-ga`

Chakrabarti, S. and Kumar, P. (2009). Test-the-REST: An approach to testing RESTful web-services. In *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD '09. Computation World:*, pages 302 –308.

Chakrabarti, S. K. and Rodriquez, R. (2010). Connectedness testing of RESTful web-services. In *India software engineering conference (ISEC)*, pages 143–152, New York, NY, USA. ACM.

Christensen, J. H. (2009). Using restful web-services and cloud computing to create next generation mobile applications. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine.

Josuttis, N. (2007). *SOA in Practice: The Art of Distributed System Design*. O'Reilly Media, Inc.

Khare, R. and Taylor, R. N. (2004). Extending the representational state transfer (rest) architectural style for decentralized systems. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 428–437.

Klein, U. and Namjoshi, K. S. (2011). Formalization and automated verification of RESTful behavior. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 541–556.

Liu, S., Liu, Y., André, É., Choppy, C., Sun, J., Wadhwa, B., and Dong, J. (2013). A formal semantics for complete UML state machines with communications. In Johnsen, E. and Petre, L., editors, *Integrated Formal Methods*, volume 7940 of *Lecture Notes in Computer Science*, pages 331–346. Springer Berlin Heidelberg.

Marinos, A., Wilde, E., and Lu, J. (2010). HTTP database connector (HDBC): RESTful access to relational databases. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 1157–1158, New York, NY, USA. ACM.

Meng, J., Mei, S., and Yan, Z. (2009). RESTful Web Services: A solution for distributed data integration. In *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*, pages 1–4.

OMG (2011). OMG unified modeling language, superstructure version 2.4.1. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF.

Pautasso, C., Zimmermann, O., and Leymann, F. (2008). RESTful web services vs. "big" web services: making the right architectural decision. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 805–814.

Porres, I. and Rauf, I. (2011). Modeling behavioral RESTful web service interfaces in UML. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1598–1605.

Rauf, I. and Porres, I. (2011). Designing level 3 behavioral RESTful web service interfaces. *SIGAPP Appl. Comput. Rev.*, 11:19–31.

Rauf, I., Ruokonen, A., Systa, T., and Porres, I. (2010). Modeling a composite RESTful web service with UML. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ECSA '10, pages 253–260.

Reza, H. and Van Gilst, D. (2010). A framework for testing RESTful web services. In *Proceedings of the 2010 Seventh International Conference on Information Technology: New Generations*, ITNG '10, pages 216–221.

Richardson, L. and Ruby, S. (2007). *RESTful Web Services*. O'Reilly Media, Inc., 1st edition.

Schreier, S. (2011). Modeling RESTful applications. In *Proceedings of the Second International Workshop on RESTful Design*, WS-REST '11, pages 15–21.

Utting, M., Pretschner, A., and Legeard, B. (2011). A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*.

Xu, D., Xu, W., and Wong, W. E. (2007). Automated test code generation from UML protocol state machines. In *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE'07)*.