Migrating Legacy Applications:

Challenges in Service Oriented Architecture and Cloud Computing Environments

Anca Daniela Ionita
University "Politehnica" of Bucharest, Romania

Marin Litoiu
York University, Canada

Grace Lewis
Carnegie Mellon Software Engineering Institute, USA

Information Science
REFERENCE

Managing Director: Lindsay Johnston
Editorial Director: Joel Gamon
Book Production Manager: Jennifer Romanchak
Publishing Systems Analyst: Adrienne Freeland
Development Editor: Monica Speca
Assistant Acquisitions Editor: Kayla Wolfe
Typesetter: Erin O'Dea
Cover Design: Nick Newcomer

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Migrating legacy applications: challenges in service oriented architecture and cloud computing environments / Anca Daniela Ionita, Marin Litoiu, and Grace Lewis, editors.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-4666-2488-7 (hardcover) -- ISBN 978-1-4666-2489-4 (ebook) -- ISBN 978-1-4666-2490-0 (print & perpetual access) 1. Systems migration. 2. Service-oriented architecture (Computer science) 3. Cloud computing. 4. Software maintenance--Management. I. Ionita, Anca Daniela, 1966- II. Litoiu, Marin. III. Lewis, Grace A.
QA76.9.S9M45 2012
004.67'82--dc23

2012026466

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 13
Considerations of Adapting Service-Offering Components to RESTful Architectures

Michael Athanasopoulos
National Technical University of Athens, Greece

Kostas Kontogiannis
National Technical University of Athens, Greece

Chris Brealey
IBM Canada, Canada

ABSTRACT

Over the past few years, we have witnessed a paradigm shift on the programming models and on architectural styles, which have been used to design and implement large-scale service-oriented systems. More specifically, the classic message-oriented and remote procedure call paradigm has gradually evolved to the resource-oriented architectural style, inspired by concepts pertinent to the World Wide Web. This shift has been primarily driven by multifaceted functional and non-functional requirements of Web enabled large-scale service offering systems. These requirements include enhanced interoperability, lightweight integration, scalability, enhanced performance, even looser coupling, and less dependence on shifting technology standards. As a consequence, several, and sometimes antagonistic, architectures, design patterns, and programming paradigms have emerged on a quest to overcome the constantly expanding enterprise software needs. In the context of resource-oriented architectures, the Representational State Transfer (REST) architectural style has gained considerable attention due to its simplicity, uniformity, and flexibility. More specifically, the potential for scalability and loose coupling, the uniformity of interfaces, and the efficient bridging of enterprise software systems with the Web are significant factors for software architects and engineers to consider REST when designing, implementing, composing, and deploying service-oriented systems. These issues stir discussion among academics and practitioners about how to properly apply REST constraints both with respect to the development of new enterprise systems and to the migration and adaptation of existing service-oriented systems to RESTful architectures. In this chapter, the authors discuss issues and challenges related to the adaptation of existing service-oriented systems to a RESTful architecture. First, they present the motivation

DOI: 10.4018/978-1-4666-2488-7.ch013

behind such an adaptation need. Second, the authors discuss related adaptation theory, techniques, and challenges that have been recently presented in the research literature. Third, they identify and present several considerations and dimensions that the adaptation to REST entails, and the authors present frameworks to assess resource-oriented designs with regard to compliance to REST. Fourth, the authors introduce an adaptation framework process model in the context of enterprise computing systems and technologies, such as Model Driven Engineering and Service Component Architecture (SCA). Furthermore, they discuss open challenges and considerations on how such an adaptation process to REST can be extended, in order to yield systems that best conform to the REST architectural style and the corresponding REST constraints. Finally, the chapter is concluded with a summary and a discussion on the points raised and on some emerging trends in this area.

INTRODUCTION

During the past decade, service-orientation has become the dominant computing paradigm in the domain of enterprise software systems. More specifically, it has been argued that Service-Oriented Architectures (SOAs) provide significant benefits to organizations, and generally allow for better alignment of business needs and IT solutions. The fundamental principle is that SOAs organize functionality as collections of interoperable services with standardized interface specification and description methods. Furthermore, service communication is independent of implementation and infrastructure allowing thus, for heterogeneous systems to communicate effectively, and for lowering costs related to integration and interoperation. Even though, SOA as a set of architectural principles is not bound to any specific technology, W3C's Web Services technologies and standards have been the primary choice of architects when implementing SOAs, especially for enterprise and B2B systems. For example, it has emerged as de facto standard that Web Service components are described by specifications written in the Web Service Description Language (WSDL), and invoked by utilizing the SOAP family of protocols. These service components usually follow one of two binding styles: Remote Procedure Call (RPC) and document-based message exchange. The RPC binding style explicitly references the service operation to be invoked, while document-based binding style promotes the usage of messages that include schema-based elements. These schema-based elements are interpreted by the receiver (i.e. server) in order to dispatch and invoke the appropriate operation. An advantage of this approach is that it is easier to validate the requests since there is a direct reference to a schema to which the message is supposed to conform to. As a result, although RPC was considered to be the dominant style when Web Services were first introduced, tools and frameworks started to support a document-based invocation style. However, in practice has been proven that the inclusion of the operation's name in the message is quite significant for the interoperability and consequently, many document-based Web Service frameworks adopted and implemented a special pattern of document-based style called the "wrapped document" pattern. In the "wrapped document" pattern the message that is sent, is essentially a schema element that is named after the operation's name, and the message that is received is also a schema element whose name follows a similar operation-based convention. In this context, a procedure is known to the service consumer, and it is readily addressable and available to be invoked by service messages, regardless of the particular binding or even technology the service may use to expose its functionality. Services that are published following the procedure-oriented Web Services stack of protocols are integrated over the Web and not through the Web—making

the term "Web" included in their name rather unfortunate. Organizations are realizing the need for a more natural integration of their systems with the Web and through the Web in a way that would not have to overcome the challenges that the Web Services stack of protocols sets. Such a development would enable the potential that is raised by exposing existing pieces of software or data as common Web resources so that, the conventional service-providing usage will become easier, and serendipitous re-usage of the resources will be possible, following the example of Web 2.0 technologies such as, mash-ups and widgets.

Resource-orientation, on the other hand, introduces the concept of content-driven decomposition of service capabilities into resources that capture and convey information. Resources are usually defined as "things" that can be named, have state, share a common, uniform interface, are visible, and possibly manipulable through representations of their state. Also, resources are associated with universal identifiers and are addressable by accepting clients' requests that, depending on the communications protocol may include control data (that is, information on how to understand and interpret the client's request), resource metadata, representation data and metadata. Consequently, resources are conceived as information-rich, stateful conceptualizations that not only provide data and functionality but also, link to each other according to specific structural and operational relationships. REST, which stands for Representational State Transfer, is an architectural style containing a set of constraints that can be used to build network-based, resource-oriented architectures. Architectures based on REST demonstrate several significant properties for distributed applications such as scalability, simplicity, reusability and performance, to name a few (Pautasso & Wilde, 2009; Pautasso, Zimmermann, & Leymann, 2008; Vinoski, 2008; Al Shahwan & Moessner, 2010). The largest example of resource-oriented architecture is the Web itself, and its architectural success attracts significant attention from the software engineering community onto how REST could be adopted in the enterprise software domain as well.

Apart from the desired properties and contextual architectural choices, traditional service-oriented systems on the one hand, and REST-based service systems on the other, demonstrate an obvious conceptual mismatch in offering service capabilities. Specifically, the procedure-oriented approach of providing software services differs significantly from the resource-oriented approach due to the distinct methodology of decomposing and publishing service capabilities as addressable units of functionality and data. In the first case, units of functionality are organized into services and service operations, which not only have specific process semantics but also, allow for data retrieval and manipulation. In the latter case, service capabilities are modeled as stateful resources and functionality is published through interaction (retrieval and manipulation) of service consumers with such content-rich resources. Furthermore, an additional diversification between procedure-oriented and resource-oriented services is that the latter makes less out-of-band or a priori assumptions, regarding the client's knowledge of an application's intra-service protocols and conventions and consequently, rely more on the understanding of common and most often standardized processing models and resource relationships. It should be noted that REST's target are Web-scale architectures that span multiple domains and the decisions made regarding the design of such systems should be based on the effect that each decision has in a network-level scale. For example, in an environment like the above, generality is preferred over efficiency in the components' interfaces. Actually this particular choice is formalized by REST with its Uniform Interface constraint. The uniformity of interfaces imposed by this constraint essentially promotes the level of independence between the communicating parts with regard to their internal technological or architectural evolution, reducing the coupling to

a minimal set of commonly accepted agreement points. Such interface constraints do not usually restrict architects that employ procedure-oriented approaches, where arbitrary operations are defined to encode custom, component-specific semantics.

It could be noted that during the last few years the community is intensively working on bridging the gap between being able to develop truly conformant large-scale implementations and the currently available methods, models and tooling. More specifically, specialized programming models, development environments, languages, and models along with infrastructure frameworks are required for organizations and businesses to be able to widely adopt RESTful approaches for designing and implementing new resource-oriented service-offering enterprise systems. However, the need for resource-orientation and alignment with the Web goes beyond new systems that are implemented from scratch. Existing service-offering enterprise systems are products of significant investments, and most often provide mission critical, well validated functionality, to a variety of clients. Redeveloping existing functionality in order to follow a more resource-oriented approach and better align service-offering procedural components to RESTful environments would include significant costs of redesigning, re-implementing and re-testing such systems as well as, maintaining duplicate functionality when required. Furthermore, fully migrating already deployed components to REST-based ones would break existing clients, which is usually not acceptable as a choice for large organizations. These issues highlight the need for a methodology to enable an automated or semi-automated adaptation of existing service-offering components in a non-intrusive manner to REST-based exposure of their functionality. In this context, there are interesting questions that arise regarding how someone may map arbitrary, domain, or business-specific procedural interfaces to actions that belong to a uniform interface across a set of resources (which is also unknown beforehand in an adaptation process and should be also specified), and to whether two paradigms as diverse as the above may converge so that existing components provide their functionalities utilizing both paradigms. In this chapter, we discuss issues and challenges that exist in this domain and we provide a holistic view of a roadmap for adapting procedure-based service-offering components to a resource oriented architecture style. In this respect, we present an adaptation framework along with a process model of how to facilitate and significantly automate the process of providing RESTful expositions of procedural functionality.

This chapter's scope and focus is to provide a baseline and a roadmap for researchers and practitioners to consider, while attempting to address the SOA/WS to REST adaptation problem. Specifically, it aims to provide and discuss the fundamental challenges pertaining to the adaptation problem and present a high-level model as a conceptual guidance for a systematic approach in addressing static, dynamic, and deployment-related adaptation concerns. As a roadmap for the adaptation concepts, that are discussed in detail in Section 3 of this chapter, we first examine the management of quality characteristics that procedure-oriented implementations of service-oriented systems demonstrate in the context of an adaptation/migration process, and how they relate to the proposed adaptation approach. Once the focus of the approach is clarified in the functional and architectural aspect, we proceed discussing issues related to the conformance of a resource-oriented architecture to REST architectural constraints, as these are defined in the REST specification by Fielding. The purpose of such discussion is to present the means by which we can assess the degree of conformance of a migrant/adapted architecture to the REST constraints and to present existing approaches that can assist a software engineer in deciding or evaluating the level of RESTfulness that the adapted view of the system should demonstrate. Since, the adaptation process has to be considered within the context of a practical implementation methodology and deployment

scenario, Section 3 discusses the principle of Model-Driven Engineering (MDE) and Service Component Architecture (SCA). MDE allows for the necessary infrastructure to represent software artifacts as MOF-compliant models and the programmatic manipulation of such models for the purposes of adaptation. Similarly, SCA provides a rich framework whereby service-oriented systems can be specified as models and their interactions can be represented in a way that can be customized in the form of different bindings. The interesting implication is that a software architect can include new bindings (such as bindings for supporting RESTful interactions with the service) and the SCA runtime will provide the necessary infrastructure for the new binding to be ubiquitously deployed and used. The interested reader who may want to embark on such an adaptation project can also delve into technical papers presented in various IEEE, ACM, and other venues, conferences, and workshops for obtaining more insights of the various low-level technical challenges involved. A collection of such related approaches is presented and discussed later in the chapter (Section 2.2).

The rest of the chapter is organized as follows. In Section 2, we summarize the background of resource-orientation, the theory of REST and RESTful services in practice, in order to provide to the reader the necessary context and for understanding the basic principles, constraints, and practical considerations of RESTful architectures. Furthermore, Section 2 discusses and presents related work with regard to specific approaches to RESTful service modeling and the adaptation problem, as these are found in the related literature. In Section 3, we present and discuss a set of considerations related to the proposed software adaptation process, as outlined above. In Section 4, we present an adaptation framework as a roadmap to gradually meeting REST's requirements by addressing REST's constraints. Limitations of the approach as well as open issues of the adaptation problem are discussed in Section 5 and a summarization of the chapter along with future research directions are presented in Section 6.

BACKGROUND AND RELATED WORK

REST in Theory and in Practice

REST in Theory

REST is an architectural style defined by Roy T. Fielding in his dissertation in 2000 (Fielding, 2000). Modern Web's scalability, flexibility, and robustness are often attributed to Web's general conformance to the REST style. However, REST is not tied to any particular standard or protocol and there are no such direct references to Web's technologies or standards for its definition. Fielding's dissertation describes several architectural styles along with the examination of induced properties and he derives REST by combining such styles. Specifically, he describes how each constraint affects architectural elements and what properties are expected to be induced when the constraints are applied in coordination. The specific constraints included in the REST specification are: Client-Server, Stateless, Cache, Uniform Interface, Layered System, and the optional Code-on-Demand constraint. The first three constraints were applied to the Web since its early architecture, while the next three were formalized and applied as the Web architecture evolved. Additionally, the Uniform Interface constraint is regarded as a central feature in REST. Brief descriptions of REST's architectural constraints are provided in Table 1.

RESTful Web Services

For a system's architecture to be fully RESTful it should conform to all of aforementioned REST's constraints. In this respect, utilizing HTTP and URIs to offer services through a Web API does

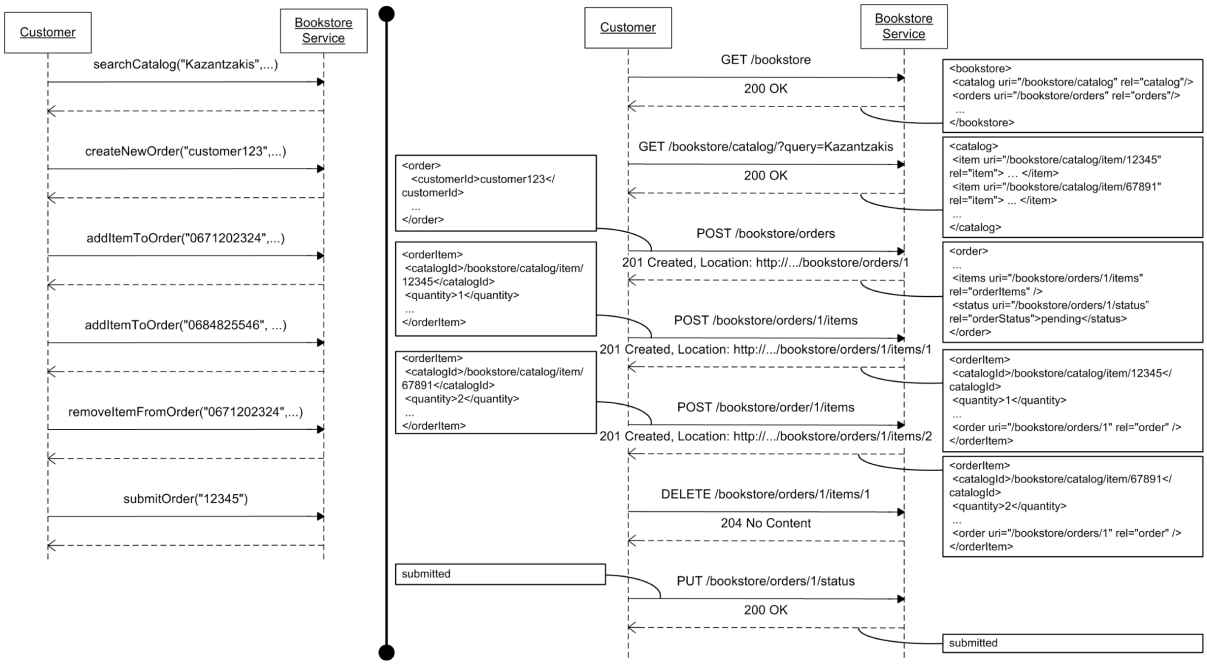
Table 1. REST architectural constraints

Architectural constraint	Description
Client-Server	The Client-Server constraint models the interactions and separates the role of requesting and the role of providing service.
Stateless (communication)	Stateless constraint refers to client-server communication and requires that every request from the client to the server is independent from previous ones. Consequently, there is no server-side session state kept during such interactions and each request should be descriptive enough to be fully understood on its own.
Cache	The Cache constraint mandates that responses by the server should indicate (probably implicitly) whether they can be cached or not.
Uniform Interface	The Uniform Interface constraint imposes the generality of components' interfaces and requires that these interfaces have system-wide universal semantics. Also, this constraint introduces and describes the resource-orientated modeling of a system's content, realized by its dependence to further architectural constraints, usually referred to as Uniform Interface's subconstraints. Specifically, REST states that the uniformity of interfaces in a RESTful architecture is obtained by its conformance to the four following constraints: identification of resources, manipulation of resources through representations, self-descriptive messages and hypermedia as the engine of application state (usually referred to as HATEOAS or the "hypermedia constraint").
Layered System	The Layered System constraint mandates that the organization of the system follows a hierarchical, layered fashion, where each layer provides services to the layer above and consumes services from the layer below.
Code-On-Demand	Code-On-Demand allows for client agents' logic to be extended by downloadable and executable code. This last constraint is an optional constraint and Fielding argues that it should be supported by an architecture conforming to REST in the general case. However, there may be contexts that this behavior is disabled and that possibility should be acceptable.

not necessarily mean that REST is applied, since HTTP offers a variety of features that may or may not be used in accordance to REST. Additionally, RESTful architectures may be implemented using any communication protocol other than HTTP, as long as it would allow for conforming to REST's constraints. Having said that, the fact that HTTP is inherently REST-enabled and, HTTP's client and server implementations are widely deployed and adopted, make this protocol a very popular and, presumably, a sound choice for implementing systems that are supposed to conform to REST. Figure 1 provides an example of a bookstore service offered through both a procedure-oriented and a resource-oriented API. The left hand side of the picture depicts the use of a procedural service API of a bookstore service where a customer is able to search a catalog, create new orders, add and remove items from orders and submit orders. These operations are directly mapped as "procedures" that pertain to services offered by a service-oriented architecture infrastructure. In this classic service-oriented paradigm, services are invoked by name using appropriate parameters. On the right hand side of the figure, the same scenario is illustrated but at this time is based on a resource-oriented architecture. In such a context, instead of services there are resources such as "bookstore," "catalog," "order collection," "order item" and "order status." These resources are manipulated using standard HTTP operations such as GET, PUT, POST and DELETE. For example, to create a new order item resource, a POST request can be issued from the client to the server pertaining to the orders collection resource. Similarly, to update the status of a bookstore order a PUT request can be issued to the order status resource.

The subset of Web-based service systems that truly follow REST principles are called RESTful Web services (Richardson & Ruby, 2007). These services utilize HTTP and URI along with common Internet's media types and Web's standards such as XML and JSON for data formatting. Furthermore, during the last few years the com-

Figure 1. Bookstore service example: procedure and resource-oriented alternatives



munity has established several conventions into using such Web's protocols and standards to facilitate the systematic development of RESTful Web services. These conventions relate for instance to the mapping of HTTP's methods ("verbs") to CRUD-like semantics (for example, using HTTP's POST method to create a new resource, GET to read a resource, PUT to update an existing resource, and DELETE to delete a resource), restricting the broader semantics that these methods have according to HTTP specification. Similarly, terms like ROA (Resource-Oriented Architecture) (Richardson & Ruby, 2007) and WOA (Web-Oriented Architecture) (Gail, Sholler, & Bradley, 2008) were introduced and defined in order to help the community better organize and communicate the concepts and the variations that REST-based designs demonstrate. Furthermore, several REST-inspired SOA patterns have been proposed (Balasubramanian, 2008), and significant research is also being conducted in the area of RESTful service composition (Pautasso, 2009).

Software Adaptation and Related Work

Software adaptation has been proposed as a discipline over the past few years (Brogi, Canal, & Pimentel, 2006; Canal, Murillo, & Poizat, 2004, 2008); however, the problem of adapting existing software components is an area of long research and discussion (Kell, 2008). Software adaptation relates to the challenges that emerge when reusing existing software artifacts in new applications and which can be addressed by introducing a category of special computational component elements called adaptors. These adaptors are responsible for enabling the communicating components to interact effectively, overcoming mismatches that may exist on both functional and non-functional aspects. These mismatches may relate to any of the four interface levels such as signatures, behaviors, non-functional properties, and semantics. Additionally, adaptation must be non-intrusive and, automatic or at least semi-automatic. In addition, the need for software adaptation is regarded

as independent of the point in the life cycle that the system may be in and emerge at any stage of its life cycle.

In order to characterize an adaptation process three parameters can be used: (a) the time that the need for adaptation is detected (requirements, static, dynamic), (b) the adaptation management (manual, automatic), and (c) the adaptation content (functional, non-functional). In this respect, the adaptation approach discussed in this chapter can be categorized as a software adaptation approach that is non-intrusive (existing component implementation is not modified and keeps offering services to existing consumers), the adaptation need is realized at the maintenance stage and the process model demonstrates a high level of automation, limiting user involvement to mostly providing declarative refinement feedback.

Apart from the practitioners' interest in being able to expose existing functionality in a RESTful manner, the challenges and issues that the transition of existing systems to resource-oriented architectures have also attracted significant attention in the academic world. The adaptation approach presented in this chapter was designed and developed by generalizing, abstracting and extending existing methods and techniques in the area of mapping procedure-oriented service-offering systems to resource-oriented ones (Liu, Wang, Zhuang, & Zhu, 2008; Laitkorpi, Koskinen, Systa, 2006, 2009; Athanasopoulos & Kontogiannis, 2010; Upadhyaya, Zou, Xiao, Ng, & Lau, 2011; Kennedy, Stewart, & Jacob, 2011). Although different in several aspects, our framework demonstrates similarities with earlier contributions to the problem, and focuses on a systematic approach of decomposing the adaptation problem into sub-problems pertaining to REST's architectural constraints and to possible architectural decisions that can be taken by the user for driving the adaptation process.

In Laitkorpi et al. (2006) authors introduce a UML-based approach to abstract legacy APIs into a canonical interface model that can be used to expose REST-like services. Their approach works on the interface level of the legacy system and above. Specifically, they regard as input an API documentation with sufficient information to run the analysis. This information is assumed to involve a set of UML models that describe the structural as well as behavioral aspects of the API. However, such models are not usually available in practice and they would require considerable effort to create them from scratch -probably comparable to writing the adapter code manually. The process proposed in that paper is split into three basic steps: a) API analysis in order to extract an API architecture, which is performed manually, b) canonicalization in order to move from the API architecture to a canonical interface model, and c) operation and structure mapping to generate the adapter code. Similarly, a subsequent work from the same group (Laitkorpi, Selonen, & Systa, 2009) describes a model-driven process for gradually transforming procedure-oriented specification models (e.g. a Sequence Diagram of top-level components) to resource-oriented interfaces. More specifically, the authors describe a process of analyzing and processing functional specifications to create an information model of the service. This model is then mapped to a resource model, which in turn is translated into RESTful service specification artifacts. Both approaches demonstrate how UML can be utilized in a model-driven process to facilitate the process and allow for model-based transitions from procedural conceptualizations of service capabilities to resource-oriented ones.

Furthermore, in Liu et al. (2008), the authors propose a process for reengineering legacy systems to REST. The process starts by analyzing the source code of the system. Informative entities driven methodologies are then used to extract candidate resources. Rules and experts' operations are applied to refine the resource list, and URIs are designed and generated based on mapping strategies. URIs may also carry information with regard to scope, resource representation, and

even business rules. Then methods are assigned and representations are designed. Finally, legacy services are wrapped by mappings to REST-based interactions. The starting point of the analysis is the source code as well as models such as ER diagrams, UML diagrams, requirements, and documentation, implying significant human involvement. Furthermore, the process is focused on the design and refinement of URIs in order to map relationship, action and other semantics.

Similarly, in Athanasopoulos and Kontogiannis (2010), a technique for identifying resources from legacy service descriptions is presented. In the adaptation technique presented, WSDL files are analyzed in order to extract REST-like resources. The technique works on the interface level using as input the machine-readable description of the service. Initially a model that captures signature information of the operations is built from the WSDL description. Then the model is extended by categorizing its elements and rules are applied to extract potential resources. Next, a rule-based resource selection is applied and the dependencies between the actual resources are captured in dependency graphs. Finally, resource identifiers are produced based on resources dependencies and operations are assigned according to patterns present on the signature model.

More recently, in Upadhyaya et al. (2011), the authors present an approach and a prototype for migrating Web Services based on SOAP, to REST-based services. The authors describe several steps for the migration process including: the identification of similar operations through clustering, the identification of the resources utilizing operation names as well as input and output parameter names, the identification of resource methods by attempting to map HTTP verbs to existing operations and finally, the message conversion between SOAP-based and HTTP messages. Their prototype also allows the user to review and possibly refine the output of the techniques before deploying the service wrapper.

Finally, viewing the problem from the client-side, the authors in Kennedy, Stewart, and Jacob (2011) discuss a protocol adapter so that SOAP-enabled clients could be used to invoke RESTful services, taking advantage of all of the Web's optimizations and especially caching. The authors provide a discussion around the problem and present a wizard-like prototype that can help the user drive the protocol adapter generation in a user-friendly manner.

Here, in accordance with most of the aforementioned approaches we also propose a model-driven approach for gradual analysis and transformation of software artifacts. Additionally, we extend the collection of the concerns that relate to this type of software adaptation, and pertain to REST constraints and features of RESTful services, and we introduce a clear separation of concerns between structural (static) and behavioral (dynamic) concerns. Then, we propose a process that aims to organize the addressing of these concerns, the capture and description of the required input, as well as, intermediary and output artifacts, and the user's involvement, so that a systematic adaptation can be achieved.

ADAPTATION CONSIDERATIONS

Service Quality Issues

Statelessness and Transactionality

Stateful communication with services is an important issue in service-oriented computing that has to be addressed before applying any adaptation technique. It should be noted that REST's requirement for statelessness affects the communication and not the service itself (resources are inherently stateful entities). Specifically, REST requires that no request should be dependent to a previous one in order to be understood and interpreted. In this respect, server-side volatile session state should not exist in a RESTful system (or at least, should

not be observable) and messages should be self-descriptive, completely indicating how they can be understood and interpreted independently for any previous interaction.

Since preserving non-persistent session state is not acceptable in RESTful architectures, a challenge that emerges in the adaptation of SOA/WS systems to REST is how transaction semantics can be modeled effectively in a truly RESTful manner. The general approach that has been proposed in the literature is designing an appropriate resource model that can model transaction semantics persistently when required. Along this lines, a specification for supporting atomicity in REST-based distributed transaction scenarios with coordinated outcomes (such as the two-phase commit protocol) is proposed by the REST-* initiative through defining transaction coordinator and transaction participant resources. Nevertheless, whether REST can accommodate or, whether it is generally suitable as an architectural style for supporting distributed transaction models has been a topic of long debate (Little, 2009; Pardon & Pautasso, 2011). In this respect, several proposals in the literature are exploring transactions and REST through introducing a variety of transaction models and techniques (Marinos, Razavi, Moschoyiannis, & Krause, 2009; Razavi, Marinos, Moschoyiannis, & Krause, 2009; Da Silva Maciel & Hirata, 2009, 2011; Pardon & Pautasso, 2011). The incorporation of such methodologies in an adaptation process may require extensive and probably intrusive reengineering of the adapted system and may fit better in a more generic migration effort—not one leading to encapsulation of existing implementations that this chapter is focused on.

General QoS Features

Most existing Web 2.0 RESTful services have usually relaxed requirements with regard to QoS features when compared to enterprise service scenarios. WS* QoS specifications usually demonstrate a high level of sophistication; however, they are often significantly complex and this is usually why they are not widely adopted. Discussions around RESTful enterprise systems with such requirements advocate a careful analysis of the requirements' rationale and goals and, the utilization of existing Web technologies as means of fulfilling them. To date, the application of standardized QoS frameworks has not been widely examined and only a few such initiatives and respective artifacts exist. This fact may be attributed to a prevailing view in the REST community that the simplicity and generality of RESTful HTTP implementations is a significant advantage that architects should try to preserve, even in complex business scenarios—solutions to business-level problems should not be technical. We consider that an architect should first investigate whether the QoS requirements for the system can be achieved in the context of a RESTful architecture and whether the properties expected to be induced by applying REST, according to its definition, are compatible with the adaptation objectives.

An adaptation approach like the one discussed in this chapter focuses mostly on the functional characteristics of the interfaces and the adaptation process is centered on resource-based exposure of service capabilities. More specifically, the major concern of such an adaptation process is the exposure of source system's functionality as a set of artifacts that define corresponding RESTful interfaces (e.g. collection of resources, media types, universal actions, hypermedia, etc.). Even though QoS requirements for the system are important and should be taken into account in a migration effort, general QoS features can be considered as concerns that go beyond the functional translation of the service interface and could be separately addressed, with the exception probably of statelessness and transactionality as discussed above. In this respect, quality characteristics can be regarded as being configurable in the underlying technology level, similar to SCA's methodological, independent treatment of

QoS aspects. In this way, certain quality features can be achieved by appropriate configuration choices of the underlying run-time component. For example, a configuration point would be whether to utilize HTTP or HTTPS for meeting a security requirement. Having said that, it should be noted that there are other quality characteristics of service-oriented systems, whose preservation is either under question with most of the existing implementation technologies and standardization efforts of RESTful services, or may, in general, require extensive reengineering of the system in order to be supported in the adapted RESTful view of the system. Such issues are still open research challenges in the community.

Levels of RESTful-ness and Induced Properties for the Target System

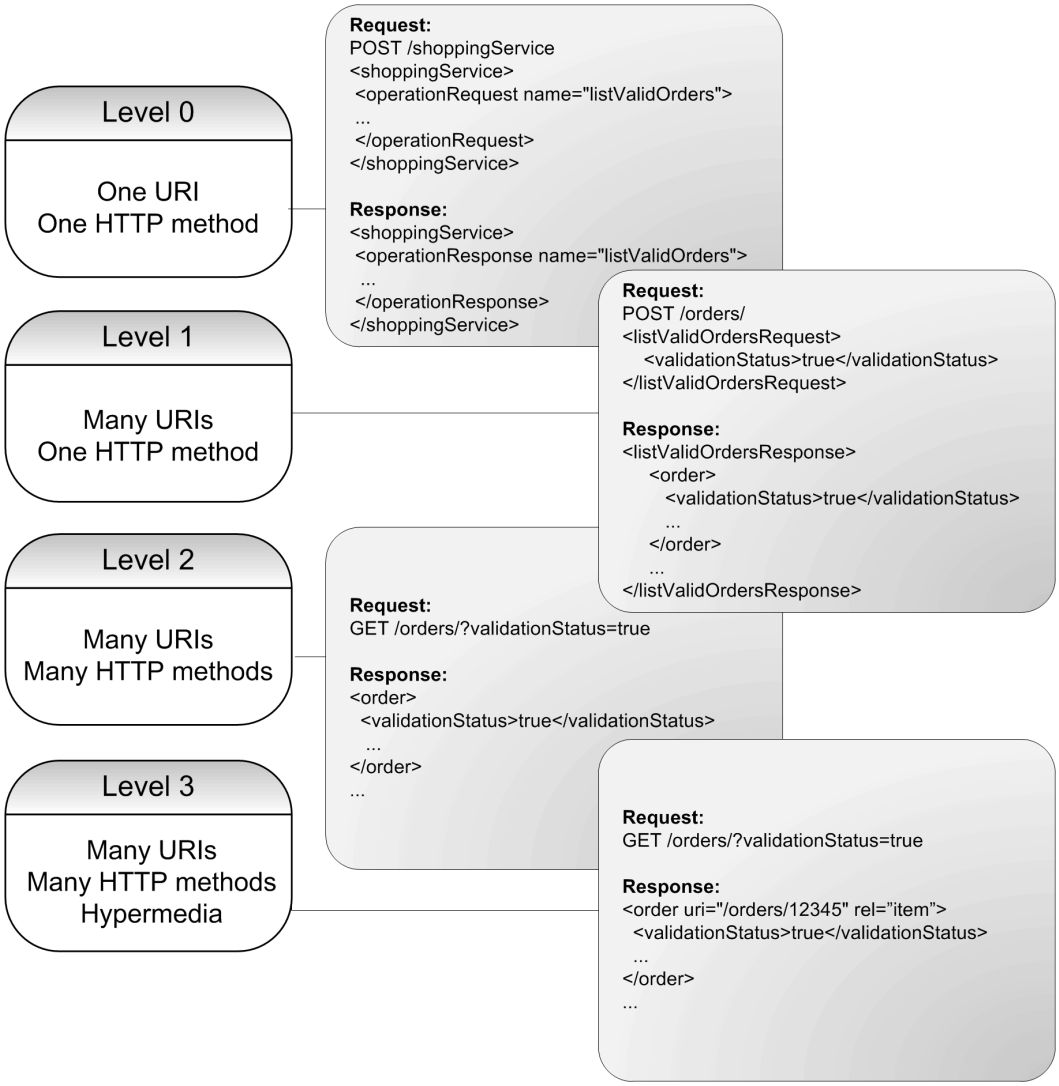
Evaluating the conformance of REST-claiming systems to REST's constraints is significant in two aspects. First, REST is an architectural style and as such, is used to convey certain architectural properties of interest, facilitating the communication and understanding between software architects, designers, and developers. Characterizing systems as being RESTful while they are not (which is a quite common case on existing Web APIs), may lead to misinterpretations among parties involved in the software development process and, eventually create misconceptions with regard to what REST really means. Second, REST includes a coordinated set of constraints, meaning that when these constraints are applied together, certain properties are expected to appear in the architecture. When one or more constraints are relaxed, probably due to certain, weighted architectural decisions that address specific issues for an application, then this deviation should be able to be captured systematically, so that the trade-offs that are included can be examined with regard to the system-wide desired properties. In this respect, REST is not a good fit for all applications and alternatives or compromises will always be present in practice. Evaluating RESTful-ness by examining the conformance of an architecture to REST's constraints assists architects in making better decisions with regard to patterns and practices used. Such conformance analysis of a design to REST constraints is important for both developing new systems and adapting existing components to REST. For this purpose, we discuss below a collection of models, approaches and techniques that have been proposed in the literature as means to assist software engineers evaluate the degree of conformance of the target adapted architecture to the REST constraints.

Evaluating Maturity and Constraint Conformance

Since there may be different architectural choices for a system, a collection of levels of maturity (or levels of conformance to REST) have been proposed. These levels of maturity of existing HTTP-based service systems with regard to REST have been empirically organized in a model presented by Leonard Richardson, the so-called Richardson's Maturity Model (RMM) as referred to by Martin Fowler (2010). RMM has four levels, each of which essentially represents different degrees of conformance to REST's constraints, and mainly to the Uniform Interface constraint. Figure 2 presents the RMM levels and depicts respective examples of HTTP-based interactions. The goal of each interaction is to retrieve a list of order items that have been validated that is part of a Shopping service.

Starting from Level 0, the HTTP protocol, although it is an application-layer protocol, is used as a transport mechanism, mainly for invoking remote procedures. At this first level, service systems usually offer a single URI as service end-point which consumers use to send messages to the server that manages the URI to be processed. The Level 0 example in Figure 2 demonstrates a single service endpoint that receives an invocation call for the "listValidOrders" op-

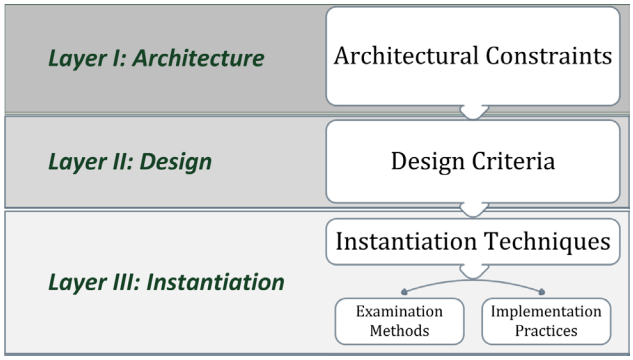
Figure 2. Levels of Richardson maturity model with examples



eration using the POST HTTP method and returns a list of order items included in a generic envelop structure. The messages usually contain structured data in formats like XML or JSON, with or without protocols like SOAP encapsulating the data. XML-RPC, XML-JSON and WS* Web Services over HTTP are typical examples of such services.

Level 1 is the first transition to a more RESTful approach by decomposing single endpoints to multiple ones, which provide semantically distinct functionality and data within a service. These "endpoints" are identified by and accessed through different URIs. In other words, Level 1 introduces the usage of REST-like resources as a way to model and expose service functionality. Up to this level, HTTP methods are not necessarily used according to their semantics and HTTP is mainly used as a medium to tunnel requests rather than as a way to convey the intent of interactions between client and server. In Figure 2, the Level 1 example illustrates that "orders" are assigned a separate URI and the invocation call is targeted

Figure 3. Abstracted view of uniform interface conceptual framework



towards that URI (POST is still used as the HTTP method for the interaction). The response includes an indication of the invoked functionality and a list of order items.

Level 2 introduces the usage of HTTP methods (or verbs) according to their semantics in order to convey to the server (and probably intermediaries) the purpose of the request. To generalize this concept, Level 2 includes services that utilize HTTP's control data to indicate the semantics and the properties of the interaction (to the extent that this can be done using a predefined set of control data). For example, GET is used for the retrieval of representations and the safety property (i.e. there should be no server-side side effects because of the interaction) that the HTTP specification requires for GET, is respected. Usually, CRUD-based services are created up until this level for the manipulation of data-rich resources. The Level 2 example illustrates the use of a GET instead of a POST, and the use of a URI to identify the proper collection of resources that the specific interaction operates upon, that is the valid orders. The response contains a representation of the resource, which is a list of valid order items.

Level 3 refers to the hypermedia constraint. Servers provide hypermedia elements to guide clients as to which are the possible future communication interactions, directing thus the transitions of the application state. Level 3 of the maturity model is regarded as a precondition to REST—but not the only one, since several more constraints must be fulfilled to meet full conformance. Level 3 goes a step further by introducing the usage of hypermedia elements (i.e. links, forms, and controls). In the Level 3 example depicted in Figure 2, in addition to the use of a GET verb and the correct use of a URI to identify the resource, the response from the server also contain hypermedia elements that can be used by the client to correctly interpret the response and plan for the next interaction.

RMM apart from indicating the maturity levels with regard to REST, it also summarizes categories of Web services that are developed under the prism of conflicting forces and essentially represent trade-offs that architects have to make, in order to induce properties to their architectures that may partly differ from the ones induced by REST. With respect to RMM and related discussions on addressing the issue of examining interface uniformity there is work that proposed a conceptual framework for evaluating and assessing a service interface against REST's uniform interface constraint (Athanasopoulos, Kontogiannis, & Brealey, 2011). The Uniform Interface Conceptual Framework (UICF), which was proposed, models a layered approach in constraint conformance evaluation, an abstracted view of which is depicted in Figure 3.

Specifically, architectural constraints in the first layer have direct reference to REST's defini-

tion according to Fielding (2000), while design criteria in the second layer constitute practical interpretations of these architectural constraints but in a technology-neutral way. These criteria often represent compromises or conventions after meticulous argumentation over how to implement abstract architectural concepts in order to obtain a uniform interface without becoming context or technology-specific. Currently, the UICF's design layer includes interpretations extracted through reviewing the literature and publications on REST organized in a set of criteria that cover a significant spectrum of issues that a REST designer faces. The major differentiating feature of design criteria at the second layer of the proposed framework, when compared to architectural constraints of the first layer, is that it lowers the level of abstraction by introducing a set of identifiable, concrete practice-oriented conceptual units in order to guide or assess design in a way that is technology-agnostic while not being technology-ignorant. Finally, the design criteria of the second layer are manifested as instantiation techniques in the third layer. In this respect, the instantiation layer is populated with realization-level configurable techniques, which can be used to either examine the conformance of an interface to the REST architectural style, or guide the implementation of systems in order to conform to it.

UICF, RMM and analogous models are of special interest in an adaptation process since such a process should be flexible enough to accommodate architectural compromises based on user input, making deviations from REST constraints' requirements possible. Specifically, the user that drives an adaptation process, should be able to intervene and apply decisions that serve his/her requirements, goals or policies but that may reduce the general conformance to REST. Using assessment models, such deviations can be systematically captured and organized so that they can be further studied in terms of their effects on desired and induced properties. Two such conformance assessment approaches that help assess the effects of a reduced constraint conformance to the induced properties, are discussed in the next subsection.

REST Constraints and Architecture-Wide Induced Properties

The problem of assessing and evaluating the compliance level of a system to REST principles has been examined on the basis of the possible side-effects of constraint deviations to the induced properties of an architecture (Navon & Fernandez, 2011). The analysis is performed by utilizing influence diagrams that reflect positive and negative effects between architectural constraints/styles and properties. Such diagrams can be constructed and used, to systematically study how each level of conformance may affect the properties, and to highlight the trade-offs included in such architectural decisions.

HTTP-based APIs have been also empirically examined with regard to Uniform Interface constraint and its subconstraints by Jan Algermissen (2010), where a discussion is also provided regarding how each expected architecture-wide property is affected, given the level of conformance to Uniform Interface constraint (which essentially defines a categorization of HTTP-based URIs). The examination is performed with regard to properties such as performance (network performance, network efficiency), visibility, modifiability (evolvability, extensibility), simplicity, scalability, as well as different costs pertaining to the architecture's lifecycle (initial, maintenance, and evolution costs).

We consider that in the adaptation process, the level of RESTful-ness for the target system results from the choices that the user guiding the process, makes. We regard the above frameworks as complementary methodologies and we do not explicitly use predefined compliance levels since the adaptation framework should be able to cover a wide range of requirements. However, the architect driving the adaptation process should be aware

to the above concepts and trade-offs, and should be able to recognize the probable side effects of the choices he or she makes during the process.

Practical Considerations: Model-Driven Engineering and Service Component Architecture

Model-Driven Engineering (MDE) (Schmidt, 2006; Kent, 2002) has been proposed as a methodology that is based on creating, processing and using models to describe, develop and document software. Software models are abstractions that represent knowledge about the domain and the application, and they are used to design, develop and even to automatically generate system artifacts, such as source code. MDE technologies are regarded as an effective way to address the complexity related to the design of software systems with complex requirements. We view the SOA/WS to REST adaptation process as being "model-driven," significantly utilizing respective Model-Driven Engineering standards and technologies. Specifically, throughout the process, models that capture system, application, technology, or adaptation-specific information can be extracted, analyzed, processed, and generated.

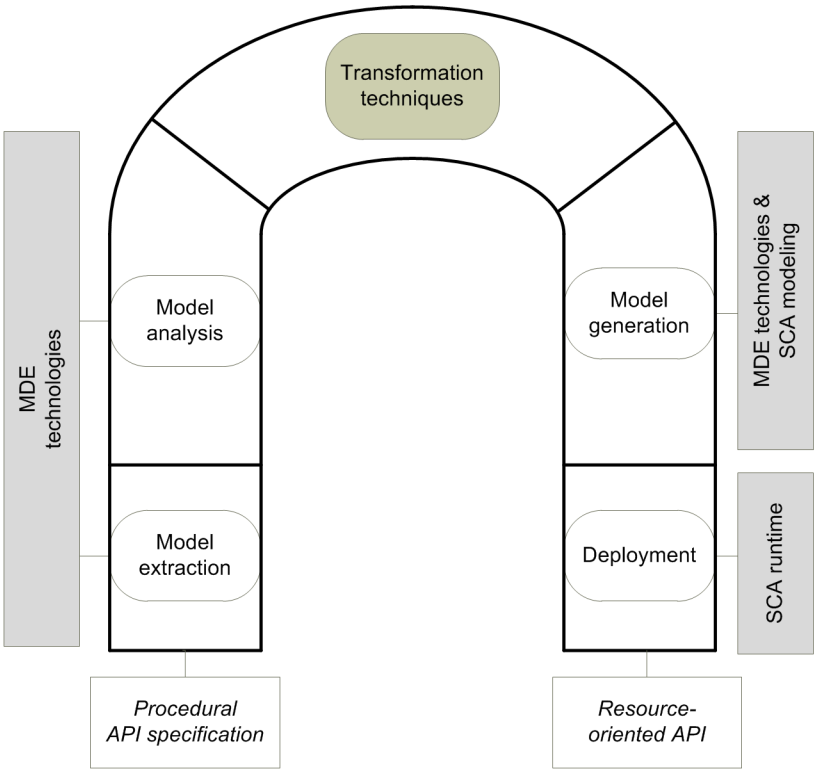
A model-based approach in building service-oriented systems and applications is proposed by Service Component Architecture (SCA). SCA is a set of specifications that uses open standards and significantly separates the concerns of non-functional requirements and service implementation assembly. SCA specifications are a product of wide collaboration in the software engineering community (Open SOA Collaboration), and since 2007, the specifications are in the process of formal standardization through OASIS (Open CSA member section). In addition, SCA provides a domain of research and several contributions on service-oriented computing research are based on or are extending SCA notions (Chu, Shen, & Jiang, 2009; Li, Zhang, & Jin, 2009). In the context of SCA and model-based assembly of service-oriented architectures, the need for converging procedure-oriented and resource-oriented components on the interface level becomes critical. In a typical SCA assembly scenario, an SCA composite is created by connecting together SCA components that provide and consume services. These SCA composites can be also used as SCA component implementations in other assemblies. Each component provides services whose interfaces are restricted in that they should be translatable into WSDL (although the actual translation may never occur in practice) with respect to the exposed functionality. The access mechanisms for the exposed services are separately handled by the definition and usage of different bindings (e.g. SOAP Web Service, JMS, EJB Session Bean, JCA, JSON-RPC, etc.).

Due to the SCA's dependency on the concept of operation, the introduction and usage of RESTful Web services in SCA assemblies is not as easy as it would be expected for such an assembly model. Similarly, the usage of procedure-oriented service systems through RESTful exposition and bindings becomes quite cumbersome. Workarounds that have been proposed include the implementation of additional source code, to manually encapsulate capabilities or the annotation of existing implementations with REST-specific tags. Both cases require considerable effort from the analyst, whose primary focus should be the composition of business functionality instead of technical issues such as infrastructure code or implementation annotations. Implementations in an SCA assembly may be based on a variety of technologies, written in different languages, and supported by different frameworks, which make intrusive workarounds less efficient. Our approach aims to enable SCA infrastructure and its runtime so that REST bindings can be added in a more flexible way, achieving thus the goals and objectives of the adaptation process. More specifically, SCA brings significant flexibility in building service-oriented architectures, and for reusing existing services (e.g. legacy services implemented in COBOL) by

assembling them together with new services that utilize modern technologies and programming paradigms. In this respect, RESTful exposition of procedural systems would provide additional benefits to organizations reusing and exposing their well-validated, value-proven systems to wider audiences, and even the Web, in a Web-friendly manner. We consider that SCA and SCA runtime environments can provide an important role in automating the adaptation process and deploying the RESTful adapted services in a unified and transparent way. For example, by utilizing a model-based framework such as SCA and by adding a new REST binding, one could access back-end system and services in a RESTful way, without losing, through the SCA runtime, the capability of accessing the same services with all the other bindings defined for this component/service.

Figure 4 depicts MDE's and SCA's roles in the adaptation framework we are proposing. In this figure, we borrow the idea of the "horseshoe" model from the area of software reengineering (Byrne, 1992; Bergey, Smith, Weiderman, & Woods, 1999) and adapt it to abstract and simplify the adaptation process we discuss in detail in the following sections, and also to highlight MDE's and SCA's involvement. In a nutshell, the left part of the horseshoe model relates to the analysis of the original procedurally-invoked service components and the extraction of possible domain models from service descriptions and data schemas. The top part of the model deals with the identification of resource descriptions and actions (i.e. create, read, update, delete) to the identified resources, given the existing functionality. The right part of the model relates to the generation of SCA infrastructure aiming to add REST bindings to existing SCA service components and SCA assemblies so that, service related resources that have been identified can be accessed in a RESTful manner.

Figure 4. MDE's and SCA's roles in the adaptation framework



ADAPTATION PROCESS AND FRAMEWORK

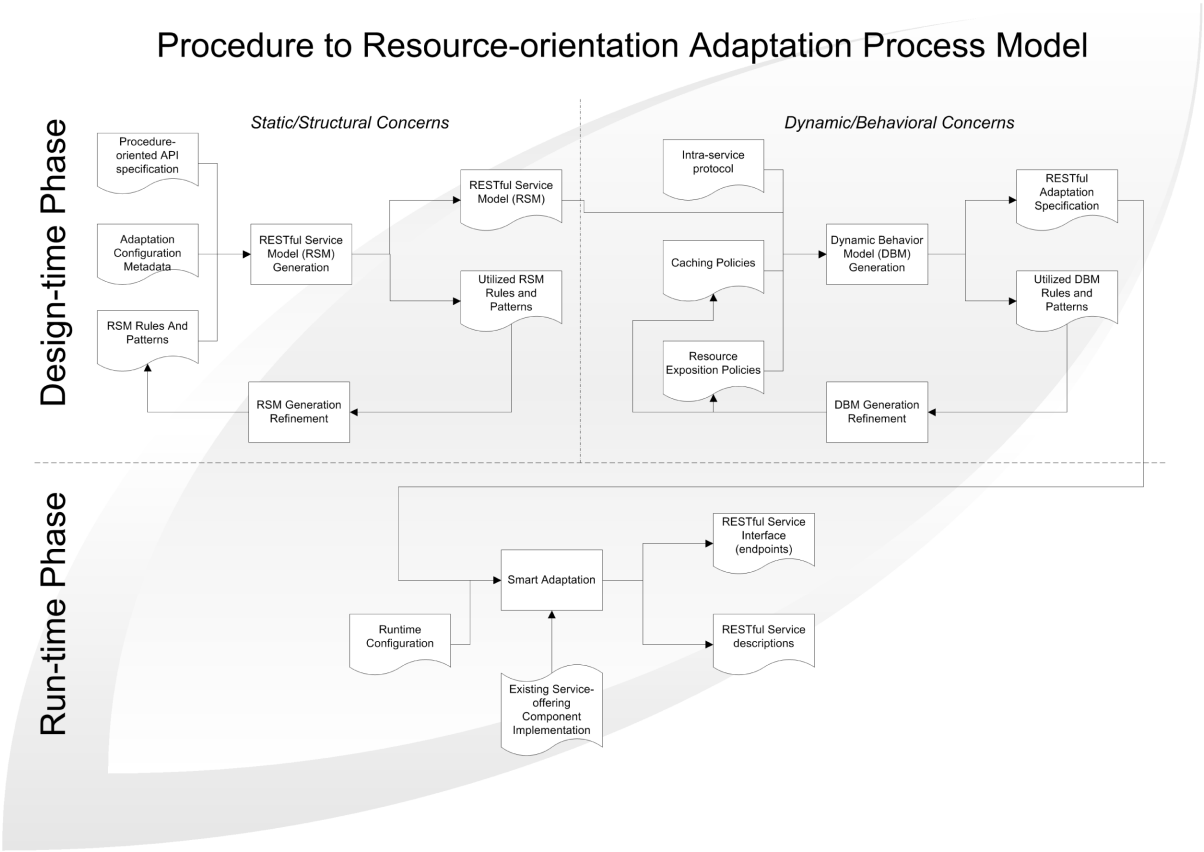
As discussed above, the non-intrusive adaptation of procedure-oriented service systems to REST-based ones requires a methodology and a corresponding process model that addresses the complete set of the required constraints included in REST. In this respect, we propose an adaptation framework working on the interface level and above, having as general target the ability to restrict the description of the system according to each constraint, in a systematic manner. Consequently, the framework entails components, which implement adaptation steps that are part of the overall adaptation process, and each of which addresses specific concerns regarding the set of REST's constraints. First of all, the adaptation process is divided into two phases based on the "time" of the application of the adaptation task: design-time phase and run-time phase. The design-time phase attempts to adapt the facets and align the views of the system's data and functionality to most of the principles of the Uniform Interface constraint (referred to as static/structural concerns), and also caching, hypermedia and certain additional interface rendering issues (referred to as dynamic/behavioral concerns). Run-time phase addresses Client-Server, Layered-System, and Stateless communication constraints. The goal of the adaptation is that the external view of the system may eventually conform to REST by respecting the restrictions each REST constraint imposes to the architecture. However, whenever this is not possible or there are trade-offs related to a full conformance to REST, the adaptation process should allow for user refinement and tuning. In this respect, the adaptation outcome should transparently allow for RESTful interactions with the service-offering component (at least to the extent that the architect decided to go). In this section, we present a model for the adaptation process along with the intermediate artifacts, to serve as a roadmap for further research to automate the vertical architectural adaptation of "RESTifing" a procedural service system. Figure 5 demonstrates the process model of the adaptation framework.

Process Model: General Description

As discussed above, the adaptation process is split into two phases: the design-time phase and run-time phase. During the design-time phase a set of techniques are applied to the specification and description artifacts of the system along with adaptation configuration metadata. User involvement is modeled in the form of output inspection and review, which essentially depends on the sophistication of the employed techniques as well as, on user's interest in applying specific architectural decisions (e.g. relaxing a certain constraint in the context of an identified architectural trade-off). The run-time phase consumes the outcome of the design-time phase and does not require user involvement other than the configuration of the run-time context (e.g. artifacts required for an SCA runtime environment). During the run-time automated mapping and management of interactions take place, which can be tuned to different levels of "smartness" given the adaptation goals (for example, managing virtual resources that represent process instance semantics and do not exist in the back-end).

The design-time phase is conceptually further split into sub-phases based on a separation between static/structural concerns of API modeling and dynamic/behavioral ones. In the first sub-phase, the process takes as input the existing API description and adaptation configuration metadata, it then applies several specialized techniques, and finally, renders a RESTful service model as an output, along with feedback data that can be edited by the user to refine the techniques' application. Specifically, during this step, resources are extracted from existing interface description artifacts and they are organized into a model based on their properties

Figure 5. Adaptation framework process model



and relationships. Next, the second sub-phase takes place, where the RESTful service model along with a dependency model of the service operations and user-defined policies about caching and resource exposition is submitted to a process that creates the final adaptation specification. This output incorporates instructions about several dynamic aspects of a RESTful service.

The RESTful adaptation specification is then used as input in the run-time phase to render an adapter that will handle requests, map them to invocations to the existing component and provide responses in a way that is compliant to REST (or to some chosen compromise). In the next section, the adaptation phases, components, and artifacts are further discussed and descriptions of each adaptation component and input/output artifacts are provided.

Adaptation Framework Components

In this section, we discuss in more detail the adaptation process as this is depicted in Figure 5.

Procedure-Oriented API Specification

We regard the adaptation process as taking place on a per service level where the term "service" represents a set of one or more operations (also referred to as procedures or functions), offered by the service-providing component in order to accomplish one or more specified tasks, and/or to provide functionality and data to service consumers. In this respect, a procedure-oriented API specification can be any programmatic application interface description that inherently follows the procedural paradigm for functionality and data

description, according to which a server-side operation should be invoked to process the specified input, and provide computed output in predefined forms (that is input and output parameters and their types are defined, described and known before the invocation). For example, interface description languages (e.g. WSDL) and generally any formalized language for expressing operation signatures and parameter types, would qualify for such an API specification. Going a step further, different syntaxes are acceptable as long as they are WSDL portType translatable, meaning that their expressiveness could be mapped concisely to the W3C language for Web Service description.

Adaptation Configuration Metadata

- The selection of the communication protocol that should be used and possibly the RESTful usage conventions of that protocol should be employed (control metadata, resource metadata, etc.) for representational state transfer.
- The selection of the identifier mechanism that should be utilized in order to assign identifiers to the extracted resources and of the predefined rules or templates that should be followed in order to meet identifier design requirements.
- The design of the format of the exchanged messages and the representation type negotiation that should take place during interactions.

RESTful Service Model (RSM) and RSM Generation Process

The RSM generation process takes as input the aforementioned artifacts and produces a service model that ideally (with the exception of alternate, less RESTful, configurations) conforms to the Uniform Interface constraint and especially to the subconstraints: identification of resources, manipulation of resources through representations and self-descriptiveness. The generated RESTful Service Model (RSM) contains a set of extracted resources, a set of identifiers that map to each extracted resource, a set of possible manipulation actions that belong to a predefined fixed set of actions, as well as, a set of representations for these resources. For each resource, identifier and representation that are extracted, explicit mappings that trace back to the original procedural service are also captured and included in the model. RSM also contains certain structural relationships between the extracted resources. More specifically, the hypermedia constraint is not fully addressed at this point; however certain resource relationships (such as containment relationships or construction dependencies) that are extracted during the generation process in conjunction with the values of respective RSM rules would also supply enough information to inject certain hypermedia information during interactions as metadata.

As discussed above, during the generation process a set of REST-like resources is extracted. This set may also be extended with virtual resources to accommodate specific modeling and mapping patterns (e.g. model non-canonicalized procedure invocations as resources with full life cycle). Furthermore, the resource relationships are discovered based on structural and naming conventions identified at the API specification. Operations are characterized with respect to their potential semantics and properties, and the interactions are "canonicalized" to the predefined choices of the selected communication protocol and possibly the subset of these choices indicated by the conventions that the user chose to be used (e.g. HTTP CRUD-like, HTTP GET-POST-only conventions). The RESTful service model should also internally preserve the appropriate mapping information that is required to eventually construct valid invocation messages to the existing back-end component.

RSM Rules and Patterns

Aside from the configuration metadata, the generation of the RESTful service model requires the application of certain transformation rules and mapping patterns. Such rules and patterns guide and configure the resource model extraction process, the assignment of resource relationships, the mapping between operation semantics, the properties and communication protocol's control metadata and, the creation of virtual resources to accommodate for different levels of conformance to REST's constraints. These rules and patterns are selected and populated with values during the RESTful Service Model (RSM) generation process. After that, the end user should be able to edit these values and reinitiate the RSM generation process which when executed again should render a RESTful service model that is compliant to the rules and values that the user chose. Some of the challenges that this set of rules and patterns aim to answer pertain to:

- The resource modeling patterns that should be used when possible, the selection of mapping patterns that are of highest priority, and the selection of the generic/static resource relationships that are of interest and of heuristics that should be utilized to extract such relationships.
- The selection of the specific rules to utilize for characterizing operations' properties (e.g. safety, idempotency, etc.), and the heuristics that should be used for the "canonicalization" of the interaction intents as identified during the RSM generation process.

RSM Generation Refinement Process

The RSM generation process is regarded as the most significant and difficult process part to fully automate, due to the fact that interface-level information does not provide all the required information to extract significant resources and to characterize operations. Also, the apparent deflection of procedure-oriented and resource-oriented application modeling would sometimes make such a generation process ineffective, especially when working with purely command-like interfaces (e.g. one-word operations with generically typed input/output). In this respect, we assume user involvement as a way to refine the generation process by providing values to a set of rules and patterns. In this way, an effective output of the RESTful service model (mainly the set of resources, a subset of their relationships, and mappings between intents and back-end invocations) is generated, which is consequently inspected before examining further aspects, especially those of prescribing the dynamic behavior of a RESTful system. However, we regard user refinement as optional, meaning that the RSM generation process should be sophisticated enough to be able to identify a set of rules and patterns to utilize. The user would declaratively participate in the process to improve the output by refining these rules and patterns and the values of their points of variability. In this respect, the initial set of rules and patterns to be utilized by the process may as well be empty.

Above we described the steps and artifacts that are parts of the design-time phase of the adaptation. Specifically, these elements are included in the first sub-phase during which mostly static/structural modeling concerns of the RESTful service interface are addressed. During the second sub-phase of the design-time phase, the RESTful service model that was generated is being enriched in order to model and prescribe several aspects of the dynamic behavior of the system that is being adapted to the REST architectural style. The dynamic aspects we consider are: caching policies about the exposed resources, exposition choices of the generated resource set (e.g. filtering) and the effective enablement of the hypermedia mechanism to guide the application state. Caching and exposition policies are imposed by user

choices, which may be supported and validated by respective mechanisms (for example, filtering policies should be validated so that there are no conflicts between the back-end functionality that is expected to be mapped and the filtering options).

Intra-Service Protocol

IDLs and generally machine-readable procedural interface descriptions like WSDL descriptions do not usually provide information with regard to the order and the conditions that each operation of the service should or could be invoked. Consequently, WSDL-translatable interface descriptions are not adequate by themselves in order to indicate how state transitions of the extracted resource set may take place through their possible manipulations. The intra-service protocol is usually implicit or described in human-readable documentation that accompanies the service. However, the problem has been identified in several research areas (for example, automated Web Service composition, automated testing of Web Services and Web Service behavioral modeling and adaptation), and several techniques to extract such ordering or dependency models between operations have been proposed (Gu, Li, Xu, 2008; Bai, Dong, Tsai, & Chen, 2005; Bertolino, Inverardi, Pelliccione, & Tivoli, 2009). In the adaptation roadmap, we regard such information either being provided by the user or extracted through employing such techniques. The intra-service protocol is rendered in terms of the existing procedural operations. The protocol's implied dependencies are mapped to resource dependencies and links which essentially create the "engine" of application state, allowing for the injection of hypermedia elements during interactions at runtime. An indication of the expressiveness of the formalism used to describe the intra-service protocol, is its equivalence to UML 2.0 Sequence Diagrams.

Caching Policies

Cache-ability is a central concept in REST and resource-oriented architectures, both in theory and in practice. REST includes it as a constraint since responses should indicate the cache-ability of the representation they are conveying. By caching representations of resources the efficiency of the communication is improved as well as, the performance as this is perceived by the requesting end. In this respect, cache-ability is a central aspect of RESTful architectures and improves scalability by allowing system-wide caching optimizations to be applied. It is generally accepted that such optimizations are critical for network-based distributed systems, in order to be able to scale. Usually service-offering procedural components do not indicate whether the responses they provide or which parts of them may be cached and for how long, and even if they do, they usually provide such information following their own patterns or techniques. However, for a system to be REST-adapted effectively, providing a process to handle such application of caching policies is regarded as mandatory in the context of an adaptation framework reference architecture. In our conceptualization of the adaptation process, caching information in the form of policies can be explicitly supplied by the user. An alternative would be to utilize techniques that analyze dynamically generated usage data. Once caching policies are defined, they are validated and then they are attached to the final output of the RESTful service model. Caching policies may also include constraints and conditions over what can be cached and how based on run-time information (e.g. based on a particular value of an exchanged representation of another resource), remaining however protocol-agnostic.

Resource Exposition Policies

The adaptation process may have as a goal a partial description of the system in terms of REST-like

resources based on contextual conditions formed at runtime. Furthermore, additional links between the extracted resources may have to be present under specific circumstances. For example, an architect guiding the process may want to restrict the granularity of the extracted resources for a particular subset of clients and provide explicit links between particular resources under state-related conditions, aligning the system to externally imposed standards or processes. In the adaptation process model discussed here, we also consider the processing and the application of such policies by explicitly providing respective information to the dynamic model generation process. In this way, the user is able to address business or technical concerns by allowing conditional exposition and linking of specific parts of the resource set in a context-aware manner.

Dynamic Behavior Model (DBM) and DBM Generation Process

In our approach, we identify and distinguish a subset of the dynamic behaviors that may occur in a RESTful system which we recognized as critical for the adaptation process and which can be prescribed during the design-time phase. The analysis that takes place is centered on processing the application's protocol and on taking into account contextual and state-specific policies that control parts of the run-time behavior of the adapter (such as hypermedia injection, caching information, etc.). The dynamic behavior model that is generated includes information about these dynamic aspects of the system in the form of consumable prescriptions by a "smart" adapter. Being able to denote and enact dynamic behavior system models as these can be achieved by the smart adapter, essentially addresses the HATEOAS or hypermedia constraint which appears to be central when the goal of the adaptation is a truly RESTful system, as well as the REST's cache-ability constraint. In addition, the dynamic behavior model of the system is constructed taking into account resource exposition policies, which provide flexibility and better alignment of the final output of the system to the adaptation goals.

DBM Rules and Patterns

The way that the particular dynamic aspects of the behavior of the service system are modeled, is guided by respective rules and patterns. These rules and patterns follow the same paradigm with RSM rules and patterns where the user can review and adapt and customize in order to render the final model. User involvement is again modeled via a refinement loop and is optional. However, the initial set of caching and exposure policies should not be empty (unless the architect is not interested in applying caching and exposition policies). Examples of such choices are, expressions regarding what sets of resources should be cached, conditions that should be met in order for the caching information to be injected, what relationships between the resources should become visible to the client/agent at runtime and under what conditions, expressions setting the resources that should be filtered, etc.

DBM Refinement Process

At this step, the user inspects the output of the DBM generation process along with the set of rules and patterns that were utilized in order to yield the dynamic behavior model of the system. He/she is then able to modify this set by either changing the values of the variation points of the rules, or to rearrange the predefined patterns available for each aspect. These actions essentially reflect to the caching and resource exposition policies. Presumably, after the refinement the generation process should automatically render a possibly different DBM in order to meet user's expectations.

RESTful Adaptation Specification

As discussed above, the dynamic behavior model is part of the final output of the design-time phase. Essentially, RSM and DBM constitute the RESTful adaptation specification. The meta-model for this specification should be expressive enough to cover both categories of concerns (static/structural and dynamic/behavioral). The adaptation specification provides all the essential information for the runtime phase of the adaptation to take place. In other words, whatever was extracted, mapped, modeled and probably refined during the design-time phase should be included or described in the final specification, which will be used as input at the deployment of the adapting component.

Following the design-time phase, the run-time adaptation phase consumes the specification as a prescription of yielding appropriate adaptation logic for a procedure-oriented service-offering component to provide a RESTful or REST-like interface. This phase is considered fully automated, given the RESTful adaptation specification from the design-time phase, and an initial configuration of the runtime/infrastructure which is the environment into which the adapter will exist.

Runtime Configuration

The adaptation process is based on the assumption that whatever adapter may exist, there is a layer of infrastructural components that are capable of dealing with a variety of technical issues, such as providing implementations and bindings for the communication protocols to be used for the RESTful adaptation of the system. We model the run-time adaptation phase as being dependent to a configuration description that essentially addresses the issues related to integrating the adapter's deployment to the available infrastructure environment. Ideally, the infrastructure should allow for the adapter to be invisible to a potential service composition process. For instance, in SCA's environment the adapter should be working on the level of the domain runtime providing a RESTful binding in a way that is transparent to anyone that assembles a service composition. However, the runtime may have to model the adapter as a separate component, with or without indicating the component's relationship or its interaction with the existing service-offering component.

Smart Adaption Process

At this point, the RESTful adaptation specification is processed and a smart adaptation component is produced, capable of accepting and processing RESTful requests, managing resources, mapping the requests to back-end service invocations, receiving the responses from the invoked services and yielding RESTful responses that include information, hypermedia and representation metadata (e.g. caching information). Furthermore, depending on the extracted resource model, the smart adapter may also serve as origin-server for virtual resources. In terms of REST's constraints, the smart adapter preserves the Client-Server style of interaction that a RESTful architecture requires. In addition, it should be noted that the smart adapter's architecture should conform to the Layered System constraint. Consequently, the client should not be able to distinguish whether it interacts with the origin-server or with an intermediary such as the adapter.

The "smartness" of the adapter relates to the sophistication of the mediation. For example, in an ideal scenario, the client-server interactions initiate via certain entry-points that map to an initial set of resource identifiers (usually mentioned as "bookmarks"). The client-server interaction beyond the entry-point identifiers should deliver the service's functionality through client's enactment to hypermedia provided by the smart container at runtime. Consequently, the resources described by the extracted resource model should become visible and probably addressable through identifiers (or identifier construction regimens) contained into smart adapter's responses. Additionally, the

adapter is responsible for applying conditional caching and exposure policies that may include state-based, request value-based or context-based conditions.

The smart adaptation process is regarded as being fully automated with respect to the production and the configuration of the adapter, since all the required information regarding the adaptation should be already provided.

RESTful Service Interface (Entry Points) and Descriptions

The smart adaptation output is the set of RESTful entry-points available for interaction and one or more descriptions of the RESTful interface (human and/or machine-readable). A description should contain the resource model, the representation types used, and probably standardized or custom relationship semantics between resources that guide the transitions between states.

The adaptation process presented in Figure 5 depicts a more detailed view of the proposed activities and tasks to transform a procedural service-oriented API to a RESTful architecture. Even though the horseshoe model (Figure 4) aims to depict a high-level abstraction and simplification of the overall adaptation in order to highlight the relationship to existing methodologies (MDE) and frameworks (SCA), it still associates to the proposed adaptation process model. More specifically, the mapping between the proposed process model and the horseshoe model can be summarized in the following points. First, the design-phase, as depicted in Figure 5, associates to the model extraction, model analysis and the application of the transformation techniques (Figure 4) as these pertain to both structural and behavioral adaptation aspects of the API. Second, the run-time phase as depicted in Figure 5 associates to the model generation and deployment phase (Figure 4) for the target adapted system.

DISCUSSION: OPEN CHALLENGES AND LIMITATIONS

This chapter has discussed a process model for adapting procedural interfaces of service-oriented systems to RESTful architectures. However, there is a number of open research issues and challenges that need be addressed.

First, not all service-offering components, and not every service, are suitable for being adapted to offer their functionality through resources and their uniform manipulation. The uniform interface that RESTful architectures require generally reduces efficiency when compared to custom procedural interfaces (Fielding, 2000). Such efficiency may be vital for a system, and a careful examination of the problem should take place before offering a RESTful version of the system's capabilities. In this respect, an interesting, open problem is how to systematically assess which procedural interfaces are good candidates to undergo RESTful adaptation, how to identify the ones that might be in conflict with RESTful exposure of the functionality, and how to capture and evaluate such incompatibilities. It is noted that the approach discussed in this chapter can be applied once the procedural interface has been empirically evaluated as a good candidate for RESTful adaptation. Additionally, the adaptation roadmap we propose does not generally address QoS and non-functional requirements that may exist for large-scale or critical software systems. As discussed above, we regard such concerns as being treated separately and we focus on an adaptation process for the functional part of the interfaces. In this respect, a systematic process and framework for assessing API suitability for REST adaptation should also take into account QoS-related concerns, especially with regard to complex security policies (e.g. authentication, non-repudiation) that the service system should support, since currently the major technologies and frameworks that are used to implement RESTful

service systems do not provide such capabilities in a standardized fashion.

Furthermore, resource extraction is a fundamentally heuristic process. In most methods that were reviewed in the literature pertaining to the extraction of resources using existing artifacts, fundamentally depended on the active involvement of an expert/user to either manually extract or modify the extracted resource collections. We restrict user involvement on refining the output of such an extraction process. However, such restriction requires the user to be familiar with the effects of the variation points that are provided for refinement. Generally speaking, the heuristic nature of the problem is linked to the fact that REST resources lack a strict formal definition and long discussions and debates take place over what should constitute a proper resource and what should not, given the definition in Fielding's dissertation.

Intra-service protocols should essentially be reflected to hypermedia mechanisms that eventually guide state transitions in a RESTful exposure of the system. However, such protocols are not usually provided, and techniques that are used to extract them do not guarantee providing all the acceptable use cases for a service. An interesting approach in the hypermedia-enablement of existing services is proposed in Liskin, Singer, and Schneider (2011). However, further research is needed in order to minimize the required informational input.

Finally, the presented adaptation process can be semi-automated during its design-time phase, requiring user involvement for refining certain aspects of the adaptation outputs as well as explicitly imposing caching and exposition policies. Further automation may be achieved though formalizing empirical knowledge into the respective issues, both during the generation of the resource model as well as during the modeling of the dynamic aspects of the interface. In addition, service usage data may play an interesting role in configuring certain aspects of the RESTful layout of the API. Such knowledge and data could be acquired by observing and analyzing system execution as well as actual user adaptation tactics when refining rules and patterns.

SUMMARY AND FUTURE DIRECTIONS

In this chapter, we discussed the problem and challenges associated to adapting procedure-oriented service-offering components in order to yield resource-oriented interfaces through appropriate runtime encapsulation. Related work in the area was presented along with considerations regarding to the application of a systematic adaptation process. Consequently, we introduced an adaptation framework along with a process model and discussed the components, the steps, and the artifacts included in the model as well as, the context in which the framework would operate. The roadmap describes a methodology framework for adapting existing services into RESTful or REST-based APIs and assists in the direction of the convergence and interoperability of two distinct paradigms in service interface design namely, procedure-orientation, and resource-orientation. Additionally, we constraint the framework on being implementation-agnostic and focus our analysis on machine-readable interface descriptions, user-provided metadata and specific interface-level information.

The proposed adaptation process model has been applied in a case study pertaining to a variety of service descriptions obtained from the Programmable Web. More specifically, we have designed and implemented a methodology and supporting prototype tools first, for the representation of mappings between procedural and resource-oriented paradigm, second, for the automatic resource model extraction, and third for the modeling of user refinement feedback. In addition, we are currently experimenting with techniques that related to the dynamic concerns

described above and we plan to build an extension to an open source SCA runtime domain that would better serve as infrastructure for our smart adaptor component. Through our experience with implementing the steps of the process model discussed above, an interesting challenge is to maintain a balance between trade-offs that related to, from one hand, the a wide spectrum and structural variety of different service descriptions (e.g. diverse possible WSDL descriptions, data schemas), and on the other hand, restricting user involvement to a simple, declarative, and easy to perform sequence of tasks.

To our knowledge, there is limited work on addressing in an end-to-end, automated or semi-automated manner, the problem of RESTful exposure of existing procedural services. Nevertheless, the area of REST and resource-oriented architectures will ever grow larger, as the need for efficient lightweight integration of components and data considered as Web resources, increases. In this context, interesting new emerging trends in the area include the specification of various QoS properties in REST as these are pertinent to WS* protocols (e.g. WS-Security), the handling of stateful systems in a stateless architecture such as REST, the denotation of transactions and transaction semantics as these are well understood in procedural systems to REST systems, and the consistent evolution/co-evolution of REST and SOA/WS models and APIs once the adaptation process is completed.

REFERENCES

Al Shahwan, F., & Moessner, K. (2010). Providing SOAP web services and RESTful web services from mobile hosts. In *Proceedings of the Fifth International Conference on Internet and Web Applications and Services,* (pp. 174-179). IEEE.

Algermissen, J. (2010). *Classification of HTTP-based APIs*. Retrieved October 10, 2011, from http://www.nordsc.com/ext/classification_of_http_based_apis.html

Athanasopoulos, M., & Kontogiannis, K. (2010). Identification of REST-like resources from legacy service descriptions. In *Proceedings of the 17th Working Conference on Reverse Engineering,* (pp. 215-219). IEEE.

Athanasopoulos, M., Kontogiannis, K., & Brealey, C. (2011). Towards an interpretation framework for assessing interface uniformity in REST. In *Proceedings of the Second International Workshop on RESTful Design,* (pp. 47–50). ACM.

Bai, X., Dong, W., Tsai, W., & Chen, Y. (2005). WSDL-based automatic test case generation for web services testing. In *Proceedings of the 2005 IEEE International Workshop on Service Oriented System Engineering,* (pp. 215-220). IEEE.

Balasubramanian, R. (2008). *REST-inspired SOA design patterns*. Retrieved October 10, 2011, from http://www.soamag.com/I24/1208-3.php

Bergey, J., Smith, D., Weiderman, N., & Woods, S. G. (1999). *Options analysis for reengineering (OAR): Issues and conceptual approach*. Technical Report CMUSEI1999TN014. Pittsburgh, PA: Carnegie Mellon University.

Bertolino, A., Inverardi, P., Pelliccione, P., & Tivoli, M. (2009). Automatic synthesis of behavior protocols for composable web-services. In H. Van Vliet & V. Issarny (Eds.), *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference ESEC and the ACM SIGSOFT Symposium on the Foundations of Software Engineering,* (pp. 141-150). ACM.

Brogi, A., Canal, C., & Pimentel, E. (2006). On the semantics of software adaptation. *Science of Computer Programming*, *61*(2), 136–151. doi:10.1016/j.scico.2005.10.009

Canal, C., Murillo, J. M., & Poizat, P. (2004). First international workshop on coordination and adaptation techniques for software entities. In C. Canal, J. M. Murillo, & P. Poizat (Eds.), *First International Workshop on Coordination and Adaptation Techniques for Software Entities,* (pp. 133-147). Berlin, Germany: Springer.

Canal, C., Murillo, J. M., & Poizat, P. (Eds.). (2008). Practical approaches to software adaptation. *Journal of Universal Computer Science*, *14*(13).

Chu, Q., Shen, Y., & Jiang, Z. (2009). A transaction middleware model for SCA programming. In *Proceedings of the First International Workshop on Education Technology and Computer Science,* (pp. 568-571). IEEE.

Da Silva Maciel, L. A. H., & Hirata, C. M. (2009). An optimistic technique for transactions control using REST architectural style. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, (pp. 664-669). ACM Press.

Da Silva Maciel, L. A. H., & Hirata, C. M. (2011). Extending timestamp-based two phase commit protocol for RESTful services to meet business rules. In *Proceedings of the 2011 ACM Symposium on Applied Computing,* (pp. 778–785). ACM.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. (Doctoral Dissertation). University of California. Irvine, CA.

Fowler, M. (2010). *Richardson maturity model*. Retrieved October 10, 2011, from http://martinfowler.com/articles/richardsonMaturityModel.html

Gail, N., Sholler, D., & Bradley, A. (2008). *Tutorial: Web-oriented architecture: Putting the web back in web service*. Retrieved October 10, 2011, from http://www.gartner.com/id=797713

Gu, Z., Li, J., & Xu, B. (2008). Automatic service composition based on enhanced service dependency graph. In *Proceedings of the 2008 IEEE International Conference on Web Services,* (pp. 246-253). IEEE.

Kell, S. (2008). A survey of practical software adaptation techniques. *Journal of Universal Computer Science*, *14*(13), 2110–2157.

Kennedy, S., Stewart, R., Jacob, P., & Molloy, O. (2011). StoRHm: A protocol adapter for mapping SOAP based web services to RESTful HTTP format. *Electronic Commerce Research*, *11*(3), 245–269. doi:10.1007/s10660-011-9075-3

Kent, S. (2002). Model driven engineering. In M. Butler, L. Petre, & K. Sere (Eds.), *Proceedings of the Third International Conference on Integrated Formal Methods*, (vol 2335, pp. 286-298). Springer-Verlag.

Laitkorpi, M., Koskinen, J., & Systa, T. (2006). A UML-based approach for abstracting application interfaces to REST-like services. In *Proceedings of the 13th Working Conference on Reverse Engineering*, (pp. 134-146). IEEE.

Laitkorpi, M., Selonen, P., & Systa, T. (2009). Towards a model-driven process for designing ReSTful web services. In *Proceedings of the 2009 IEEE International Conference on Web Services*, (pp. 173-180). IEEE.

Li, W., Zhang, Y., & Jin, J. (2009). Research of the service design approach based on SCA_OSGi. In *Proceedings of the 2009 International Conference on Services Science Management and Engineering,* (pp. 392–395). IEEE.

Liskin, O., Singer, L., & Schneider, K. (2011). Teaching old services new tricks: Adding HATEOAS support as an afterthought. In *Proceedings of the Second International Workshop on RESTful Design,* (pp. 3–10). ACM.

Little, M. (2009). *REST and transactions?* Retrieved October 10, 2011, from http://www.infoq.com/news/2009/06/rest-ts

Liu, Y., Wang, Q., Zhuang, M., & Zhu, Y. (2008). Reengineering legacy systems with RESTful web service. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference,* (pp. 785-790). IEEE.

Marinos, A., Razavi, A., Moschoyiannis, S., & Krause, P. (2009). RETRO: A consistent and recoverable RESTful transaction model. In *Proceedings of the 2009 IEEE International Conference on Web Services*, (pp. 181-188). IEEE.

Navon, J., & Fernandez, F. (2011). The essence of REST architectural style. In Wilde, E., & Pautasso, C. (Eds.), *REST from Research to Practice*. Berlin, Germany: Springer. doi:10.1007/978-1-4419-8303-9_1

Pardon, G., & Pautasso, C. (2011). Towards distributed atomic transactions over RESTful services. In Wilde, E., & Pautasso, C. (Eds.), *REST from Research to Practice*. Berlin, Germany: Springer. doi:10.1007/978-1-4419-8303-9_23

Pautasso, C. (2009). RESTful web service composition with BPEL for REST. *Data & Knowledge Engineering*, *68*(9), 851–866. doi:10.1016/j.datak.2009.02.016

Pautasso, C., & Wilde, E. (2009). Why is the web loosely coupled? A multi-faceted metric for service design. In *Proceedings of the 18th World Wide Web Conference,* (pp. 911-920). ACM.

Pautasso, C., Zimmermann, O., & 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,* (pp. 805-814). ACM.

Razavi, A., Marinos, A., Moschoyiannis, S., & Krause, P. (2009). RESTful transactions supported by the isolation theorems. In *Proceedings of the 2009 International Conference on Web Engineering,* (pp. 394-409). Springer-Verlag.

Richardson, L., & Ruby, S. (2007). *RESTful web services*. New York, NY: O'Reilly.

Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *Computer, 39*(2), 25-31.

Upadhyaya, B., Zou, Y., Xiao, H., Ng, J., & Lau, A. (2011). Migration of SOAP-based services to RESTful services. In *Proceedings of the 13th IEEE International Symposium on Web Systems Evolution,* (pp. 105–114). IEEE.

Vinoski, S. (2008). Serendipitous reuse. *IEEE Internet Computing*, *12*(1), 84–87. doi:10.1109/MIC.2008.20

KEY TERMS AND DEFINITIONS

Architectural Style: A set of constraints on architectural elements (component, connectors, data elements), their features, their roles and their relationships applied in coordination to induce certain system-wide properties to the conforming architectures.

Model-Driven Engineering (MDE): Is a methodology based on creating, processing and using models to describe, develop and document software.

Representational State Transfer (REST): An architectural style for designing network-based hypermedia applications. REST includes six architectural constraints and was invented by Roy Fielding while developing HTTP.

RESTful Web Services: Is a collection of technologies and practices that utilize existing Web standards and protocols to develop and provide services over the Web.

Service Component Architecture (SCA): A collection of specifications that uses open standards and separates the concerns of non-functional requirements and service implementation assembly.

Software Adaptation: The actions related to producing a category of special computational component elements called adaptors in order to be able to reuse existing software artifacts in new applications without altering their implementation.

Web Services: A collection of standards and technologies to implement SOAs. Web Services use the SOAP family of protocols to exchange XML-based messages to access service functionality and return service results. Service interfaces usually include operations whose signatures and invocation mechanisms are described by WSDL documents. Web Services are centered on procedural conceptualizations of service capabilities.