Modeling RESTful applications

Silvia Schreier
Chair of Data Processing Technology
University of Hagen
silvia.schreier@fernuni-hagen.de

ABSTRACT

Today, Representational State Transfer (REST) is becoming more and more important. RESTful web services are an alternative to Remote Procedure Call technologies like SOAP and WS-* services. There are many frameworks for implementing RESTful applications, but there is still a lack of support for the early phases of the development process, particularly analysis and design. For building formal models of RESTful applications an appropriate metamodel is needed. After analyzing existing approaches and techniques a first version of such a REST metamodel is presented and used to model an example application. Beside enabling modeling, such a metamodel offers a vocabulary for REST in practice and the basis for model driven development.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design; H.1 [**Models and Principles**]: Miscellaneous

General Terms

Design

Keywords

REST, metamodel, modeling, web services

1. INTRODUCTION

Roy Fielding introduced Representational State Transfer (REST) in 2000 as an architectural style for distributed hypermedia systems [3]. An architectural style "is a named set of constraints on architectural elements that induces the set of properties desired of the architecture" [3]. REST is an abstraction of the basic architecture of the Hypertext Transfer Protocol (HTTP) and concentrates on concepts instead of syntax and technical details [14].

When developing applications for the World Wide Web (WWW) the idea of the underlying architecture and its properties should be taken into consideration. Understanding REST can help to achieve a better performance and scalability, as well as looser coupling, which enhances interoperability [14] and serendipitous reuse [19]. Due to these benefits, more and more developers are trying to apply REST to their architectures, but there are still problems and reservations against this style [15, 16, 17] which are considered in more detail in the next section. Many frameworks[1] that support the implementation of RESTful applications evolve. But tool support and best practices for the other phases of the development process, particularly application modeling, are still missing. Fielding describes the REST style at a high level of abstraction. Based on a more technical view the presented metamodel tries to build a terminology for the design and implementation of RESTful applications in practice and provides a basis for further solutions, e. g., model driven development. Such a terminology can help to understand and to enhance the development of RESTful applications.

After the related work in the next section and an introduction to metamodeling in Section 3, a first version of a metamodel for a model-driven approach and an example application are presented in Section 4 and 5. The article concludes with Section 6, which also outlines future work.

2. RELATED WORK

Much research has been done in the field of developing RESTful applications. Foundations of REST are due to Fielding [3]. Tilkov [14] and Richardson and Ruby [11] provide best practice examples and hints on how to develop RESTful applications.

Kopecký et al. [5] present hRESTS as a solution for missing machine-readable Web APIs of RESTful services. They argue that a microformat is the easiest way to enrich existing human-readable HTML documentations. They introduce a model for RESTful services, but with a focus on documentation and discovery.

Alarcón and Wilde [2] introduce a metamodel for descriptions of RESTful services which is the basis for the Resource Linking Language. They focus on service documentation and composition. In contrast to the Web Application Description Language (WADL) [4] they identify links as first class citizens.

Furthermore, Liu et al. [7] introduce an approach for reengineering legacy systems to RESTful Web Services. They outline the key issues in this area and propose a solution which covers identification of resource candidates, relation

---

[1]e. g., Jersey (`https://jersey.dev.java.net/`) and Restlet (`http://www.restlet.org/`)

and operation analysis, URI and representation design, as well as service construction. In their opinion, the key problem is to find the right granularity for the resources.

In the domain of model driven development, Laitkorpi et al. [6] propose a process for designing RESTful services that focuses on a model based identification of the resources and on generating corresponding WADL descriptions.

Most publications use their own informal model, tailored to their needs, for describing the solution. A metamodel for designing and implementing RESTful applications is still missing.

Pautasso et al. [10] compare REST with WS-* services based on architectural decisions, including the strengths and weaknesses of both variants. They mention the confusion about best practices, the missing possibility to generate client stub code for RESTful applications, and the increased maintenance effort of multiple representation formats as main weakness of REST. Additionally, they identify the lack of standardized methods for designing [10] or documenting [5] RESTful services as problem. Beside, Vinoski [15, 16, 17, 18] names the missing tool support, understanding and best practices in addition to the fact that most developer are used to programming languages which use specific interfaces as reasons why developer do not use REST.

The metamodel presented in Section 4 tries to provide a first basis to solve these issues.

3. METAMODELING BASICS

Based on Stahl et al. [12] this section gives an introduction to metamodeling as an essential part of every model driven development. A metamodel of a model specifies its structure and meaning. Without such definition automated handling or model driven development, e. g., code generation and model transformation, are impossible. A metamodel defines the *abstract syntax*, the possible elements and their relations in between. In addition, it specifies its *static semantic*, the constraints for well-formed models. Every element of a model is an instance of the corresponding metamodel element.

A *concrete syntax* for a metamodel can be defined by a textual or visual language. It is possible that more than one concrete syntax for one metamodel exists.

A metamodel can be defined using natural language but this is informal and inhibits tool support. For that a meta-model for the metamodel, a meta-metamodel, is needed. A meta-metamodel is typically described by itself.

There are two popular meta-metamodels. The Meta Object Facility (MOF) [1] by the Object Management Group (OMG) which is used as meta-metamodel for the Unified Modeling Language (UML) and Ecore [13] which is part of the Eclipse Modeling Framework (EMF) and based on Essential MOF (EMOF) [1]. As suggested by Stahl et al. [12] we choose the latter one because of its simplicity and available tool support. MOF should only be used when the UML is chosen as metamodel as well because there is only tool support for the UML metamodel and not MOF only. But the UML metamodel is not appropriate to model RESTful applications because it contains object-oriented concepts which conflict with REST constraints, e. g., the uniform interface.

Following, the needed Ecore elements and their properties are described.

EClass An *EClass* represents a concept and is identified by a name. It can define a set of *EAttributes* describing its properties and a set of *EReferences* describing its relations to other *EClasses*. It may have a set of superclasses. An *EClass* inherits all attributes and references of its superclasses. An *EClass* can be marked abstract which means that it is not allowed to create instances of it.

EDataType An *EDataType* represents primitive types like integers and strings and is identified by a name.

EAttribute An *EAttribute* is identified by a name and its type is defined by an *EDataType*. It has properties to define the lower and upper bound of how many values are allowed for this *EAttribute* and if the values are ordered.

EReference An *EReference* models a (possible) directed relationships between the instances of two *EClasses*. They are identified by a name and define the type of the relationships target. If a bidirectional relationship shall be modeled the *EReference* has a corresponding opposite *EReference*. It has a boolean flag which defines if the source of the relationship contains the target. If the span of life of the container ends, the one of the contained element also ends. Every *EClass* can be contained in at most on container. An *EReference* has properties for defining lower and upper bounds to describe the multiplicity and if the set is ordered.

4. THE REST METAMODEL

The REST metamodel is divided in structural and behavioral modeling. The former one describes the possible resource types, their attributes and relations as well as their interface and representations. The latter one offers the possibility to describe the behavior with state machines. At this time it does not support modeling representation details but its planned for future versions.

We use a web album, similar to Picasa[2] or flickr[3], as example for a RESTful application.

4.1 Structural Modeling

As a first step, the structural elements of the metamodel have been identified.

Fielding [3] mentions *resources* as one main element. Because we usually find resources with similar properties and behavior in an application grouping of *resources* can be helpful to describe RESTful applications during development. The model uses the term *resource* when talking about one concrete element and introduces the term *resource type* to describe common aspects of multiple *resources*. Because of this, we have identified the elements below including their attributes and relations between them, which are illustrated in Figure 1 and 2.

The described elements are modeled as *EClasses*, their properties as *EAttributes* and their relations as *EReferences*. The *EDataType* of *EAttributes* are omitted because they are generally obvious. Their lower and upper bound are defined as one if not specified otherwise.

---

[2]`http://picasaweb.google.com`
[3]`http://www.flickr.com/`

resource type	description
primary resource	core concepts of the modeled domain, e.g., pictures and albums
subresource	part of another resource, which should also be addressable directly
list resource	list of all concrete resources of a primary resource
filter resource	list of concrete resources with desired properties
projection resource	contains only a subset of attributes of another resource
aggregation resource	aggregates attributes of different resources to reduce the interaction amount
paging resource	splits large resources in different pages
activity resource	stands for one single step of a workflow

Table 1: Different resource types [14].

4.1.1   ResourceType and ResourceIdentifierPattern

A *ResourceType* models a concept or object and its properties. It is an abstract *EClass* and has a name. It has an attribute maxResources which specifies how many resources of this type are allowed. It can be any positive number or infinity. Tilkov [14] distinguishes between different kinds of resources (see Table 4.1.1). These types build an inheritance hierarchy, which is illustrated in Figure 1.

Because of the *uniform interface*, activities, which go beyond create, read, update, and delete, have to be modeled in a different way. If we want our photo album to provide a feature to collect excellent photos, a suitable workflow needs to be defined: pictures can be suggested and a photographer has to review them. Such a suggestion can be modeled as an *ActivityResourceType*. An *ActivityResourceType* is normally a nominalisation of an activity.

Furthermore every *ResourceType* has to contain at least one *ResourceIdentifierPattern*. A *ResourceIdentifierPattern* is abstract and can be a *SimpleIdentifier* which is described using a string or a *ComplexIdentifierPatter* which uses the values of the *ResourceType's Attributes* (see Section 4.1.2). The identifiers are not important at the beginning of the design process but at the end for implementation or code generation.

A *ResourceType* contains an unordered set of named *ResourceElements*, like *Attributes* and *Links*.

The relations to the other model elements are illustrated in Figure 2.

4.1.2   Attribute and DataType

Attributes specify a *ResourceType's* properties, can be marked as optional and conform to a defined *DataType*. A *DataType* can be a *PrimitiveDataType* or a *CollectionType*. An *PrimitiveDataType* is identified by its name, typical examples are integers and strings. Valid literals can be described using an Extended Backus-Naur Form or something similar. A *CollectionType* represents an ordered set of values, similar to variable arrays in programming languages and references the *DataType* of its contained elements.

4.1.3   Method, MethodType and Parameter

Because of the *uniform interface*, a *MethodType*, which is identified by its name, has to be defined for all existing methods, e.g., the HTTP verbs. A *ResourceType* is associated with a set of supported *Methods* which have to have a *MethodType*. The *Method* element is responsible for the behavior and determines the set of produced and consumed *MediaTypes*. Additionally every *Method* can define *Parameters* which can be contained in a consumed *MediaType* or in the resource identifier. A *Parameter* has a *DataType*.

4.1.4   Link and RelationType

Links support *hypermedia as the engine of application state (HATEOAS)* and can be marked as optional. We distinguish between *InternalLinks* and *ExternalLinks*. Each *Link* can define a media type independent *RelationType* [8]. So the client knows which kind of relation exists between the two resources and which method requests with which meaning can be send to the link target. For example the *RelationType* can contain navigation information like next or up. More examples for possible *RelationTypes* can be found in [8]. An *InternalLink* refers to one target *ResourceType*. *ExternalLinks* are necessary to model links to resources outside the current application and only define a resource identifier. Additionally, *LinkCollections* can be used to handle a set of links, e.g., all elements in a *ListResource*. They are divided in *External-* and *InternalLinkCollections*. The distinction is necessary for the later parts of the design process.

4.1.5   MediaTypes and Representations

MediaTypes enable *content negotiation* and are identified by a name. They can contain named *MediaTypeElements* which have a *DataType* and define the contained information. A *Representation* is identified by a name, defines its *MediaType* and models the data sent by the server. It consists of named *RepresentationElements* which define their *DataType* and can have a corresponding *MediaTypeElement*. The concrete appearance of a *Representation* or *MediaType* are not covered by the metamodel at this stage but could be added later on, e.g., using templates.

4.2   Behavioral Modeling

In accordance with Fielding, representations are used "to capture the current or intended state of that resource" [3]. To describe the behavior of the *ResourceTypes*, deterministic finite-state machines were chosen. This offers the possibility to represent the current resource state and to define how a resource reacts to a certain request. The concrete behavior of a method is described in an imperative way, e.g., for changing the value of *Attributes* and *Links*.

Following, the elements for modeling the behavior, which are illustrated in Figure 3, are introduced.

4.2.1   State and Transition

Every *ResourceType* defines a set of *States*. Exactly one of them is the initial state which is entered after the creation of a resource of this type. A *State* can have an arbitrary number of outgoing *Transitions*. Every *Transition* specifies exactly one target *State*. A *Transition* defines one or more *Trigger*, e.g., an *InternalMessage*. In addition, the transition can define a *Condition*. *Conditions* are not modeled in more detail in this metamodel version, but are planned for future versions. At the moment a textual description is
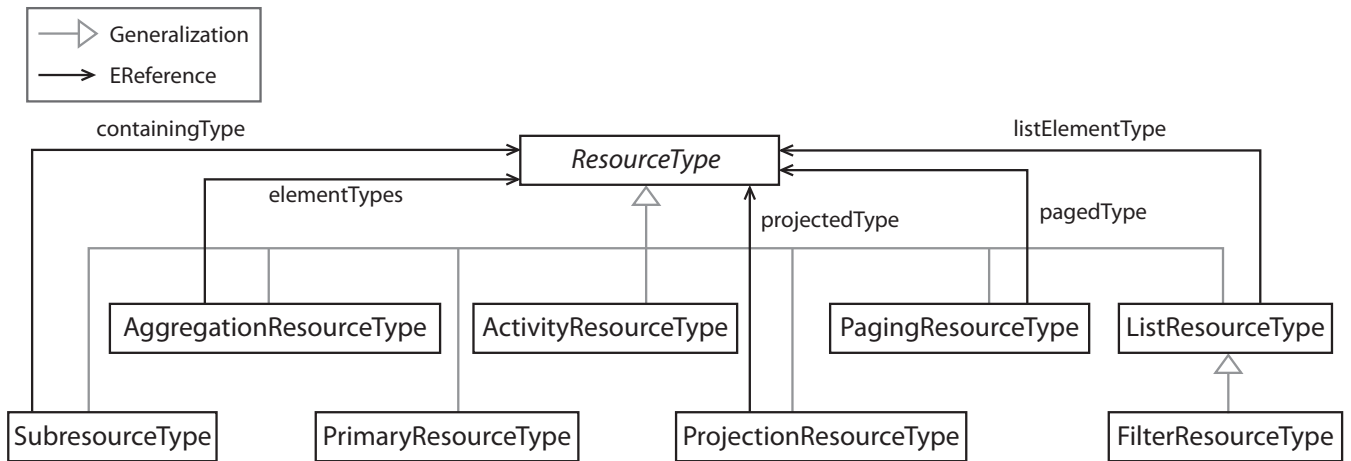
Figure 1: Hierarchy and relations of the resource types. (Details like attributes and multiplicities are omitted here for clarity.)
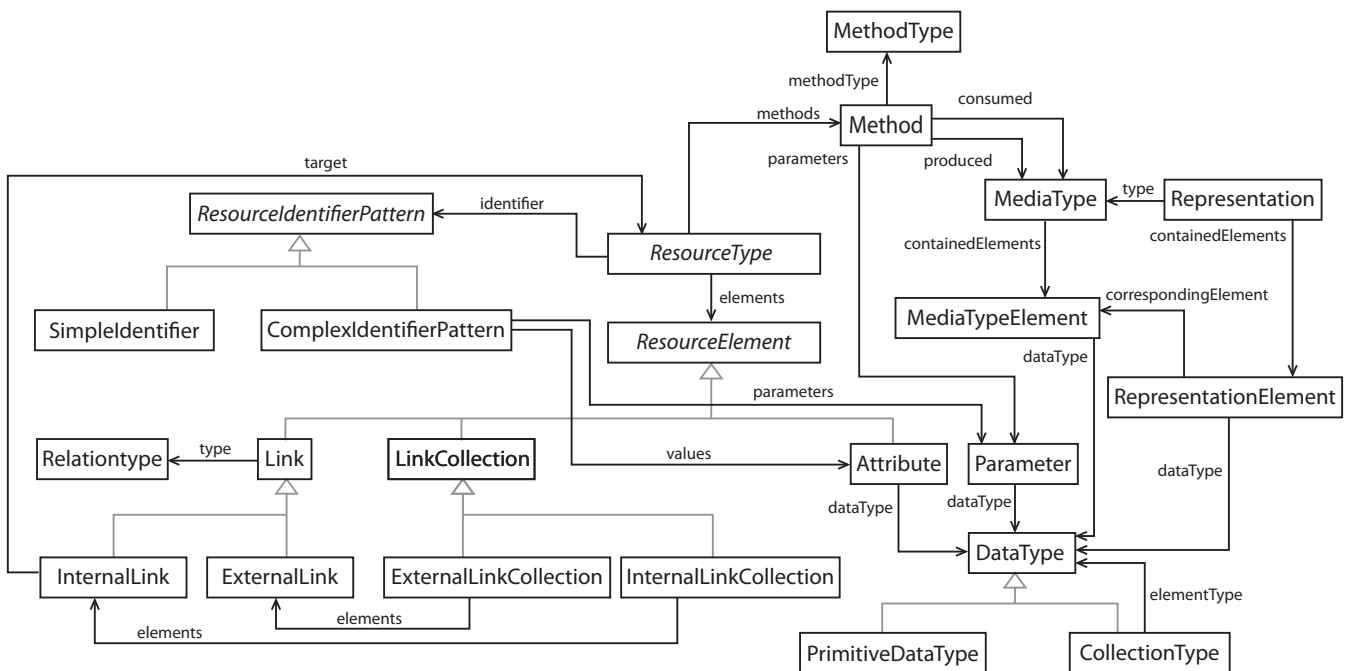

Figure 2: The core of the structural metamodel. (Details like attributes and multiplicities are omitted here for clarity.)
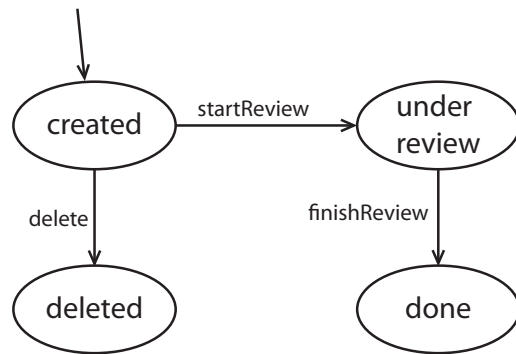
Figure 3: The core of the behavioral metamodel. (Details like attributes and multiplicities are omitted here for clarity.)

used. To add behavior, every *Transition* can define a *BehaviorSpecification* (see next section) and every *State* can define one which is executed when it is entered and one when it is left. Additionally, every *State* defines the set of supported *Methods*. If a resource is in a certain *State* and receives an incoming Message the appropriate *Method* is chosen and executed. If there is no supported *Method* with the same *MethodType* no *Method* will be executed and an adequate *StatusCode* has to sent as response.

4.2.2 BehaviorSpecification

Every *Method* needs to define its behavior. Such a specification consists of one *Action*. There are *ActionSequences* which represent a sequence of *Actions*, *CreateActions* to create new resources and *MessageActions* for internal messages or sending requests to other resources. *UpdateActions* offer the possibility to change the value of *Attributes* and *Links*. *ReturnActions* allow us to define the response, including the *Representation* and all metadata. To be able to express these *StatusCodes* and *Metadata* are part of the metamodel. Additionally, *ConditionalActions* can be used to define the desired behavior. The *ListAddAction* and the *ListRemoveAction* allow to add and remove elements of collections.

One possibility to define the semantic constraints of different *MethodTypes* is to constrain the allowed *Actions* in the *BehaviorSpecifications*. For example no *UpdateAction* would be allowed in a `GET` specification.

4.2.3 Creator

To define the creation of resources every *ResourceType* can define a *Creator*. Similar to *Methods* a creator defines *Parameters*, consumed *MediaTypes* and a *BehaviorSpecification*.

5. EXAMPLE

As mentioned before, we use a simple photo album as ex-

Figure 4: The example ResourceTypes.

ample. We begin with the identification of the *PrimaryResourceTypes*, which are: `Picture` and `Album`. We omit a `User` *ResourceType* here for clarity. Every `Album` and `Picture` has a title and a description as *Attribute*. Additionally, a `Picture` has and an excellent flag. For handling the picture file data we introduce the *SubresourceType* `PictureData` which has the binary data as only *Attribute*. In addition to these *PrimaryResourceTypes* we need *ListResourceTypes*. Every `Album` has an *InternalLink* to the list of the contained `Pictures` and a list of all `Albums` is needed. A `Picture` has *InternalLinks* to its `PictureData` and `Album` as well as to the previous and to the next `Picture`. The *ListResourceType* `PicturesOfAlbum` has an *InternalLink* to the corresponding album. For modeling the excellent picture workflow the *ActivityResourceType* `Suggestion` is needed as well as a corresponding *ListResourceType* `AllSuggestions` and a *Filter-*

Figure 5: The states of a suggestion.

ResourceType `SuggestionsUnderReview`. A `Suggestion` has a statement, the current status and the number of positive and negative reviews as *Attribute* and an *InternalLink* to the suggested *Picture*. This part of the model is illustrated in Figure 4 using a notation similar to UML class diagrams.

After that the *MediaTypes* and *Representations* and their elements have to be chosen. The concrete appearance and layout of the representations has to be defined in later steps.

Subsequently, the behavior of the *ResourceTypes* has to be defined. We choose the `PicturesOfAlbum` and the `Suggestion` type as example. The `PicturesOfAlbum` resource has only one state. A method of the type `GET` should response with a list of all contained pictures, e. g., as an Atom feed [9]. The method types `PUT` and `DELETE` are not supported by this *ResourceType*. A method of the type `POST` should create a new picture. For that we have to choose the consumed and produced media types, e. g., an Atom entry [9] containing the title of the picture. We define a *Creator* in the *Picture* resource which can consume this media type. The *BehaviorSpecification* of the `POST` method contains an *ActionSequence*, starting with a *CreateAction* and a *ListAddAction* for adding the new element to the list. After that it returns a `303` *StatusCode* and the link to the created picture resource as part of the HTTP header. The methods of the other *ResourceTypes* can be defined in similar ways.

The `Suggestion` type defines the *States* illustrated in Figure 5. In the initial state `created` the *MethodTypes* `PUT`, `DELETE` and `GET` are supported. The first one can be used to update the statement, the status or the suggested picture which can be received by sending a `GET` request. The *BehaviorSpecification* contains a *ConditionalAction* so that the `startReview` *Transition* to `under review` is triggered if the status is changed to *under review*. The corresponding `DELETE` *Method* sends an internal message to trigger the `delete` *Transition*. In that *State* a `410` *StatusCode* is returned. Th *State under review* only supports `GET` and `POST`. The first one returns the current state of the resource, the latter one allows to add a positive or negative review using an *UpdateAction*. If there are more than two positive reviews the corresponding `Picture` is marked as excellent, if there are more than two negative reviews it is marked as not excellent. This can be modeled by using *ConditionalActions*. In both cases the method triggers the `finishReview` *Transition* to the *State* `done`. If there are not enough reviews no *Transition* is triggered. The *State* `done` only supports a *GET* request.

6. CONCLUSION

This paper has presented a first version of a REST metamodel and its applications to a small example. Advanced aspects like authentication, e. g., for users, are not part of the metamodel at the moment. After defining the behavioral model in more detail, the next step toward the validation of the metamodel is the application to different case studies. Testing the metamodel with various scenarios in addition to using the vocabulary defined by it are suggested as validation techniques by Stahl et al. [12]. The lessons learned thereby will help to improve the metamodel incrementally. Additionally a textual language for models is planned. Also the analysis of RESTful applications and their implementations in different programming languages and paradigms can be used as further input. The long-term goal is to create a metamodel that allows developers to design models with a high level of abstraction and technical models. With transformation rules between these models the support of the entire development process becomes possible. HTTP as the most prominent realization of REST will be a core aspect of the metamodel. But at the same time, it shall not be mandatory so that the metamodel becomes as independent and universal as possible. After the metamodel has been improved in this way, the next step will be the development of code generators for different languages.

Especially the missing tool support and the lack of understanding mentioned by Vinoski [17] can be solved using and improving this metamodel. Furthermore a textual language can help concentrating on resources instead of the problems arising when object oriented design is mixed up with resource design during implementation. So the developer can get used to the uniform interface and other REST constraints.

7. REFERENCES

[1] Meta Object Facility (MOF) Core Specification. Object Management Group, January 2006.

[2] R. Alarcón and E. Wilde. RESTler: crawling RESTful services. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 1051–1052, New York, NY, USA, 2010. ACM.

[3] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[4] M. J. Hadley. Web Application Description Language (WADL), February 2009.

[5] J. Kopecký, K. Gomadam, and T. Vitvar. hRESTS: An HTML Microformat for Describing RESTful Web Services. In *WI-IAT '08: Proc. Int. Conf. on Web Intelligence and Intelligent Agent Technology*, pages 619–625. IEEE, 2008.

[6] M. Laitkorpi, P. Selonen, and T. Systa. Towards a Model-Driven Process for Designing ReSTful Web Services. In *ICWS '09: Proc. Int. Conf. on Web Services*, pages 173–180. IEEE, 2009.

[7] Y. Liu, Q. Wang, M. Zhuang, and Y. Zhu. Reengineering Legacy Systems with RESTful Web Service. In *COMPSAC '08: Proc. Int. Software and Applications Conf.*, pages 785–790. IEEE, 2008.

[8] M. Nottingham. Web Linking. Request for Comments: 5988. Internet Engineering Task Force (IETF), October 2010.

[9] M. Nottingham and R. Sayre. The atom syndication format. Request for Comments: 4287. Internet Engineering Task Force (IETF), December 2005.

[10] C. Pautasso, O. Zimmermann, and F. Leymann. Restful Web Services vs. "Big" web services: Making the Right Architectural Decision. In *WWW '08: Proc. Int. Conf. on World Wide Web*, pages 805–814. ACM, 2008.

[11] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, 2007.

[12] T. Stahl, M. Völter, S. Efftinge, and A. Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt.verlag, 2007.

[13] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Longman, 2009.

[14] S. Tilkov. *REST und HTTP: Einsatz der Architektur des Webs für Integrationsszenarien*. dpunkt.verlag, 2009.

[15] S. Vinoski. REST Eye for the SOA Guy. *IEEE Internet Computing*, 11(1):82–84, 2007.

[16] S. Vinoski. Demystifying RESTful Data Coupling. *IEEE Internet Computing*, 12(2):87–90, 2008.

[17] S. Vinoski. RESTful Web Services Development Checklist. *IEEE Internet Computing*, 12(6):94–96, 2008.

[18] S. Vinoski. RPC and REST: Dilemma, Disruption, and Displacement. *IEEE Internet Computing*, 12(5):92–95, 2008.

[19] S. Vinoski. Serendipitous Reuse. *IEEE Internet Computing*, 12(1):84–87, 2008.