Institute of Architecture of Application Systems

A model-driven approach for REST compliant services

Florian Haupt, Dimka Karastoyanova, Frank Leymann, Benjamin Schroth

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{haupt, karastoyanova, leymann}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{INPROC-2014-23,
  author    = {Florian Haupt and Dimka Karastoyanova and Frank Leymann and
               Benjamin Schroth},
  title     = {A Model-Driven Approach for REST Compliant Services},
  booktitle = {Proceedings of the IEEE International Conference on Web
               Services (ICWS 2014)},
  year      = {2014},
  pages     = {129 - 136},
  doi       = {10.1109/ICWS.2014.30},
  publisher = {IEEE}
}
```


Universität Stuttgart
Germany

A model-driven approach for REST compliant services

Florian Haupt, Dimka Karastoyanova, Frank Leymann, Benjamin Schroth
Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstr. 38, 70569 Stuttgart, Germany
{firstname.lastname}@iaas.uni-stuttgart.de

***Abstract*—The design of applications that comply to the REST architectural style requires observing a given set of architectural constraints. Following these constraints and therefore designing REST compliant applications is a non-trivial task often not fulfilled properly. There exist several approaches for the modeling and formal description of REST applications, but most of them do not pay any attention to how these approaches can support or even force REST compliance. In this paper we propose a model-driven approach for modeling REST services. We introduce a multi layered model which enables (partially) enforcing REST compliance by separating different concerns through separate models. We contribute a multi layered meta-model for REST applications, discuss the connection to REST compliance and show an implementation of our approach based on the proposed meta-model and method. As a result our approach provides a holistic method for the design and realization of REST applications exhibiting the desired level of compliance to the constraints of the REST architectural style.**

Keywords—REST; architectural style compliance; model-driven software development; architectural constraints; representational state transfer

I. INTRODUCTION

Representational State Transfer (REST) is an architectural style for distributed hypermedia systems formally defined as a set of constraints. These constraints have to be followed by REST compliant architectures [1]. The rationale behind this definition is that compliance with the constraints defined by REST implies a set of desirable nonfunctional properties, like for example scalability or network efficiency. Existing implementations of REST compliant architectures form a loosely coupled, scalable and fast system that can evolve further without losing these properties. Such architectures, in turn, have positive effects on the operating IT department and finally on the business supported by these systems.

The World Wide Web (WWW) is considered the one big REST system of today. It exists for a long time already; it grew from around 600 web servers in 1996[1] to over 400 billion web servers in 2012[2] and still performs. Originally it has been built for humans accessing documents but is gaining more and more adoption as platform for applications interacting with each other. For humans interacting with hypermedia documents, the REST constraints have been very well met, mainly due to the use of HTML. Considering in contrast applications hosted in the WWW, in most cases the REST constraints are not fulfilled to their full extent [2][3][4]. This leads to the fact that these applications are often not exploiting the full potential of the architecture of the WWW and the REST architectural style. As REST has been designed with long term goals like scalability and evolvability in mind [1], partly disregarding the REST constraints may have no impact in the short term, but in the long time it is expected to have a negative impact on the non-functional properties of a system.

Model-driven techniques have been proposed to improve the development of complex applications [5]. In model-driven software development (MDSD), software is not implemented manually based on informal descriptions but mainly automatically generated based on formal models. Models are first class citizens in the process of application design and realization. This approach in general leads to better code quality, fewer errors, increased reuse of best practices, better maintainability through "standardized" code, and increased portability through the separation of platform independent models (PIM) and platform specific models (PSM).

In this paper we introduce a model-driven approach for the design and realization of REST applications. Besides the described advantages of the model-driven approach in general we focus on how to achieve the goal that the created application complies as much as possible with the REST constraints. In this respect, the main contributions of this paper are (a) a set of meta-models for the design and realization of REST applications, (b) a discussion of how these models can help to create REST compliant applications, (c) an associated role model and (d) a prototypical realization of the proposed approach.

The paper is organized as follows. In section II we will discuss how the REST constraints have been realized by the architecture of the WWW and derive from this the motivation for our work. In section III we will introduce a multi layered model for REST applications, the base for our model-driven approach for the development of REST applications, and also the associated role model. In section IV we will discuss the relation between the proposed set of meta-models and the design and realization of REST compliant applications. In

[1] http://www.w3.org/2005/01/timelines/timeline-2500x998.png
[2] http://news.netcraft.com/archives/category/web-server-survey/

section V we will introduce our graphical tool supporting the proposed approach. In section VI we will provide an overview of related work in relevant areas and in section VII we will summarize our work and point out possible future work.

II. The REST Constraints and their Implementation in the Architecture of the WWW

The World Wide Web (WWW) is the biggest and best known architecture following the REST architectural style [1]. More precisely, the REST architectural style has originally been defined to document the rationale behind the architecture of the WWW. In the following we will discuss, how each constraint defined by the REST architectural style is realized in the architecture of the WWW. We will show that some of the constraints are already fulfilled by the Hypertext Transfer Protocol (HTTP) [6] while others have to be explicitly followed by application developers. The following investigation will contribute to the conclusion that there is the requirement for supporting and improving the design and realization of REST services.

The *Layered Client Server* constraint demands the separation between client and server components and prescribes a layered system structure. This constraint is an inherent part of the architecture of the WWW. It is reflected by the HTTP specification defining corresponding roles, for example client, server or proxy. The general concept of layered systems is very well known and established in different domains. This constraint can therefore be seen as fulfilled by default. Building an application for the web, i.e. an application that is used over the web, inherently implies building a layered system.

The *Cache* constraint requires that response data can be labeled as cacheable or not cacheable. Cache components can then be placed anywhere between client and server components to intercept, save and afterwards deliver cacheable data. In the WWW this constraint is realized by the HTTP protocol. HTTP defines several header fields that allow controlling the caching of response messages. In addition, the HEAD method defined by HTTP is used for the validation of stale resources.

The *Stateless Server* constraint in general has to be fulfilled by applications hosted on a server. The application developer is responsible to design an application to be stateless. Nevertheless, the concept of stateful and stateless applications has not been introduced by the REST architectural style, it is also known from other domains. In Java EE development, *Stateless Session Beans* represent stateless behavior while *Stateful Session Beans* represent stateful behavior [7]. When scaling a system based on parallelism, also known as *scaling out*, statelessness is also an important aspect [8].

The *Uniform Interface* constraint prescribes that all interactions have to be based on a fixed set of predefined and well known methods, the so called uniform interface. This constrained is directly fulfilled by the HTTP specification. HTTP defines a fixed set of methods with their corresponding semantics, for example GET, PUT POST and DELETE [6]. This set of methods forms the uniform interface of the WWW. Despite this, it is nonetheless possible to use the HTTP methods in a wrong way ignoring their predefined semantics. Therefore, fulfilling the Uniform Interface constraint also requires application developers to understand and properly use the basic HTTP methods.

Besides the Uniform Interface constraint, the REST architectural style defines four additional interface constraints. The *Identification* constraint states that resources are identifiable. This constraint is implemented by HTTP, where resources are identified using URI [9].

The *Manipulation through Representations* constraint introduces the distinction between a resource and its representation. In the WWW, this concept is again realized by HTTP. The payload of every HTTP message is typed; the type of the message body is defined by a corresponding header field using MIME media types [10]. Additional header fields for content negotiation allow one resource to provide multiple representations.

The *Self Descriptive Messages* constraint requires that a message contains all information necessary to understand the contained resource representation. This constraint is implemented in HTTP by the separation of data in the message body and metadata in the message header. HTTP uses MIME media types to indicate the type of the resource representation contained in the message body.

The *Hypertext as the Engine of Application State (HATEOAS)* constraint defines that the state of a client application is directly controlled by the resources it accesses. The representation of a resource has to contain all the metadata that is needed to know how a client application can interact with the resource and if and how it is related to other resources. The Hypertext Markup Language (HTML) fulfills this constraint using hyperlinks and forms. Hyperlinks can be used to navigate from one resource to related resources. Forms define the possible interactions with a resource; they define which data a client application may send to which resource. Nevertheless, HTML is only one possible representation format. It has been designed to present structured text and media to humans. In the context of machine to machine communication scenarios often different, mostly domain specific, representations are used. Therefore, it is the responsibility of the application developer to design representations in a suitable way. Today, the HATEOAS constraint is often not fulfilled. In contrast to all the other constraints defined by the REST architectural style, the concepts addressed by the HATEOAS constraint are new to application developers.

The WWW is *the* reference architecture for the REST architectural style. When developing a service for this platform, we demonstrated that a subset of the REST constraints is already fulfilled by standards the WWW is based on, mainly the HTTP protocol together with URI and MIME. Nevertheless, other constraints have to be fulfilled by the service itself, hence by an appropriate service design and implementation. The constraints mainly relevant in this context are the uniform interface, i.e. the proper use of the HTTP verbs, and the HATEOAS constraint. Empiricism shows, that the fulfillment of these constraints is a non-trivial task as most services on the WWW that claim to be RESTful, i.e. REST compliant, are in fact not [2] [3] [4].

To increase the proper adoption of the REST principles and to help service designers and developers to create REST compliant services we see the need for formal concepts, methods and suitable tool support for this task. In the following, we will present our model-driven approach for this challenge. We will show how model-driven software design techniques can help to observe REST constraints and to design and realize REST compliant services.

III. A Layered Meta-model for REST Applications

In this section we will introduce the set of meta-models which is the basis for our approach for the model-driven development of REST services. After giving a short overview, we will discuss each meta-model in detail. We will describe its main components, why and how it is separated from the other meta-models and how it is positioned in the context of the whole approach. We will also define an associated role model. The main meta-models as well as their interrelations are shown in Fig. 1.

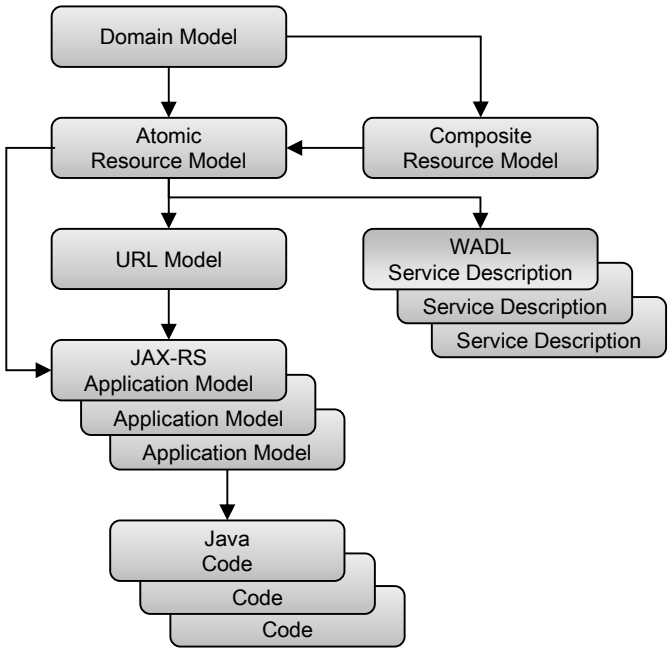

Fig. 1. Layered meta-model for REST applications

The *Domain Model* provides the possibility to model an application independent of REST, based on a meta-model best fitted to a certain application domain. The domain model is then mapped to a composite or atomic resource model. The *Composite Resource Model* allows grouping several atomic resources into one composite resource to reduce the complexity of a resource model. The *Atomic Resource Model* is the core model allowing modeling an application in terms of resources, their interrelations and the interfaces offered by them. This resource model can be transformed into an *REST Service Description* serving as an interface description for clients.

The *URI Model* defines the URI structure for the resource model, i.e. by what URI(s) each resource is identified. One important aspect of our approach is the separation of the resource model from the URI model. This is directly influenced by the HATEOAS constraint which, inter alia, demands to use links (or any other metadata embedded in the representation of a resource) to connect resources. As a consequence, a service client should not make any assumptions about a URI structure but rather rely on links and other metadata offered as part of the resource representation. This is one key aspect of REST to enable loose coupling between a service and its clients [11]. In our approach and as shown in Fig. 1 this leads to the REST service description being directly derived from the resource model and independent of the URI model.

Based on a given resource and URI model a platform independent *Application Model* can be derived. As there are many target frameworks and platforms available, our approach allows defining several application models. In the last step, the application model will be transformed into application code which can then be deployed on an appropriate target platform.

A. Domain Model

One of the main challenges in the design of REST applications is the introduction of the paradigm of resource orientation. In addition, the interaction with resources has to be solely expressed using the uniform interface, which in case of the WWW is defined by HTTP. Building a REST application requires to model both, the structural as well as the behavioral aspects of an application, in terms of resources and HTTP verbs. In contrast to this resource-oriented approach, service-oriented application design is mainly shaped by method- or object-oriented application interfaces. Especially in the field of web services the dominant interface description, the web service description language (WSDL) [12], follows a traditional paradigm by describing interfaces in terms of a set of operation comprising input- and output data.

The top meta-model shown in Fig. 1, the *Domain Model*, enables to model an application using a modeling paradigm best fitted to the application domain as well as to the roles involved in defining it. A domain expert responsible for defining an application interface may be only familiar with, for example, the concepts of object-orientation or entity-relationship diagrams. The domain model is used to define an application interface independent of REST. This allows taking advantage of existing expert knowledge without the need to introduce new modeling concepts.

After defining the domain model it will be transformed into a resource model, either a composite or an atomic resource model. This step typically involves overcoming a certain *impedance mismatch*. The term impedance mismatch has been originally shaped in the context of the mapping of object-oriented structures to relational structures [13]. It describes the fact that this mapping is non-trivial and not complete, i.e. the meta-models of object-orientation and of relational data structures are not mutually compatible. A common approach to tackle this challenge is to automate the object-relational mapping. This typically reduces the complexity of the mapping task for the user and allows avoiding errors by implementing well proven best practices and executing them in an automated manner. Nevertheless, the mismatch between object-orientation and relational data structures still exists; each mapping typically includes tradeoffs and impurity.

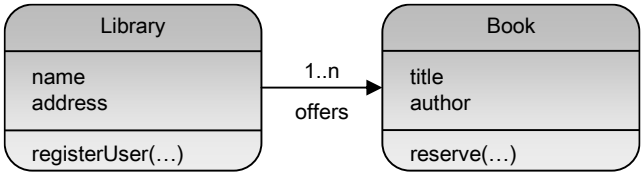

Fig. 2. Domain model example

The same concept of an impedance mismatch can be observed when mapping non resource oriented meta-models to a resource oriented meta-model. In our approach, the domain model is not prescribed; it is defined depending on the needs of the application domain. In this paper and in our realization we will assume the domain model to be based on a simplified version of the object oriented model defined as follows. The main components of the model are *entities* and *relations*. Entities contain attributes and methods and can be connected to other entities using relations. An example of a model based on this meta-model is shown in Fig. 2. A *Library* has a name and an address and *offers* a set of books. In addition, a library provides a method to register new users. A *Book* has a title and an author and offers the functionality to reserve a book for later borrowing. For reasons of comprehensibility, we omitted details like data types in this example.

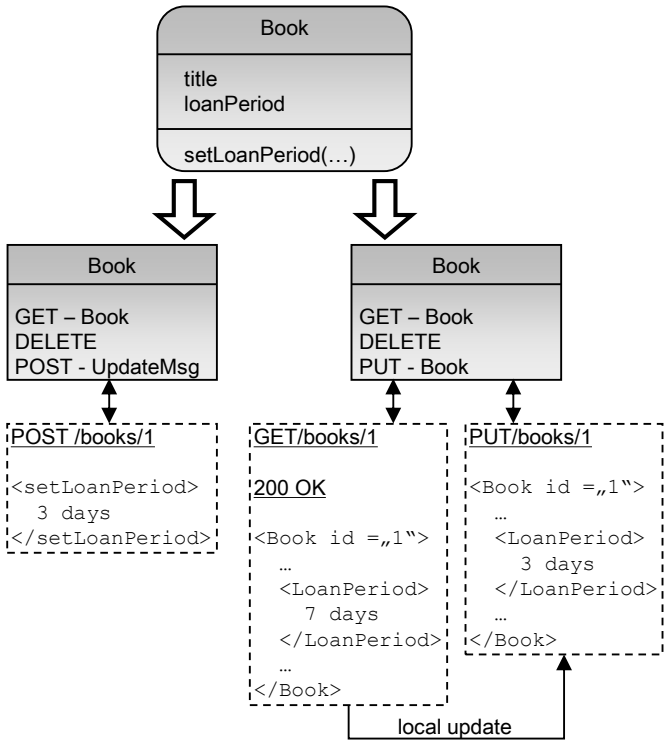

Fig. 3. Updating strategies

In the following we will demonstrate the concept of impedance mismatch based on the example shown in Fig. 3. The operation *setLoanPeriod()* updates the loan period of a book. The object *Book* will be mapped to a corresponding Book resource. The operation *setLoanPeriod()* can be mapped in two different ways. The first approach is using the HTTP PUT method and shown in the right part of Fig. 3. In contrast to the setLoanPeriod() operation, which requires only the new loan period, the PUT method demands to provide the complete resource representation. Therefore, updating the loan period requires fetching the resource representation of a book using GET, updating it locally, and writing it back using PUT. The second approach is based on the HTTP POST method and shown in the left part of Fig. 3. To update the loan period, a corresponding update request is send to the book resource using POST. There are several tradeoffs between these two approaches. The first approach based on GET and PUT uses only idempotent HTTP methods, i.e. in case of a network failure they can safely be replayed. On the other hand, it requires multiple interactions and may introduce concurrency issues. The resource may be modified by a different client between the GET and the PUT, also known as *lost update problem*. The second mapping, based on the POST method, requires only one interaction and avoids concurrency issues. As a drawback, the POST method is not idempotent, i.e. it cannot be replayed safely and also allows no caching.

By using a non-resource oriented domain model and an automated mapping to the resource model, the domain expert can simply define an abstract strategy like “prefer safe operations” without the need to define each mapping in detail, a tedious and potentially error prone task. In our approach, these mapping strategies are defined in a separate model, used as an external marker when executing the transformation from domain to resource model.

B. Composite and Atomic Resource Model

For the resource model, we distinguish between a composite resource model and an atomic resource model. The composite resource model is an additional abstraction layer that aims at supporting a modeler in the definition of complex resource structures. The composite resource model is an extension of the atomic resource model providing additional composite resources. One composite resource represents a set of interconnected atomic resources. Using composite resources as modeling constructs can reduce the complexity of a resource model and therefore helps to maintain and understand complex resource structures. As shown in Fig. 1 a domain model can be mapped to a composite resource model as well as to an atomic resource model. Similarly, when not using a domain model one can directly model a resource structure as a composite resource model as well as an atomic resource model.

An example for a composite resource and its mapping to atomic resources is shown in Fig. 4. In the upper part a composite resource representing a long running computation (LRC) is shown. Modeling resource structures for long running computations is a common task in the design of REST applications and there exist several best practices for this [14] [15]. The atomic resource model for the composite resource *MyCalculation* is shown in the lower part of Fig. 4. It consists of three resources each supporting a subset of the uniform HTTP interface. The *Manager* resource is used to retrieve a list of all existing tasks or to start new computation tasks. For each computation task, a *Task* resource is created. This resource represents one long running computation, it can be used to retrieve the current state of the computation or to modify or cancel it. For each intermediate or end result produced by a

computation, a *Result* resource is created. Results can be retrieved or, if no more needed anymore, deleted. This simple example clearly demonstrates that the use of composite resources can support the effective modeling and interpretation of resource models. The number of resources is reduced, well known structures are hidden and the automated transformation from composite resources to atomic resources promotes the use of well-known best practices in resource modeling and helps to reduce modeling errors.

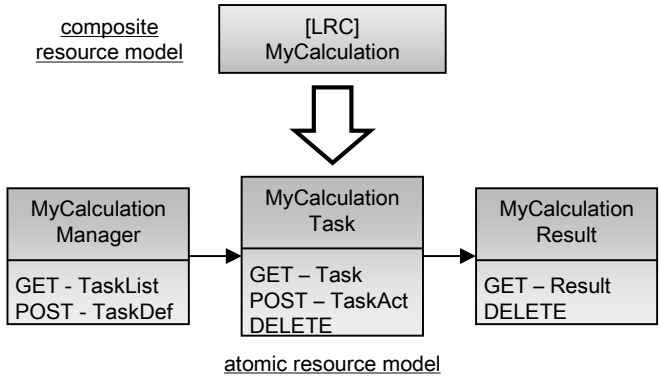

Fig. 4. Composite and atomic resource model example

C. Service Description

The *Service Description* is derived from the atomic resource model. It is basically a view on the resource model serving as an interface description for service clients. There have already been made several proposals for the formal descriptions of REST APIs; examples are WADL [16], Swagger [17], or RestDesc [18]. Our approach generally supports to generate any appropriate kind of service description, for example depending on the target audience and specific use cases. Nevertheless, most of today's service description languages cannot be generated solely from the resource model. They often mix both, a description of the resource structure as well as information about the URI structure of a REST API.

As already mentioned before, one key aspect of our approach is the separation of the resource model and the URI model. This separation helps to realize the HATEOAS constraint, demanding that any interaction is driven by the resource representations themselves without any additional information like the structure of URIs. The description of a resource oriented service interface is exclusively derived from the resource model and may also include information from the domain model. In contrast, any service description derived by this approach does not contain any information about the URI structure or any platform or implementation specific details. This increases the decoupling between client and server, one of the key aspects of REST.

D. URI Model

The REST architectural style demands to uniquely identify any resource by an URI. In our approach the resource model describes a set of resources and their interconnections. The URI model references this resource model and assigns an appropriate URI structure to it. From a client perspective the URI structure of an REST application should be irrelevant. REST compliant clients are hypertext driven, they access well known root resources and afterwards use links or any other metadata like forms retrieved from resource representations to interact with an application. Nevertheless, for the REST application provider, the URI structure does matter. On the one hand, it has to be assured that each resource is uniquely identified by at least one URI. On the other hand, a well-structured URI structure significantly helps in maintaining an application.

The URI model is separated from the application model as it is independent of the realization of the REST application described by the resource model and the URI model. The use of standards, like for example URI and HTTP, is one important driver for the increasing adoption of REST for application interfaces. The combination of the uniform interface, a standardized resource identification mechanism and the concept of resource representations achieves to keep any implementation details away from the clients of a REST application.

E. Application Models and Code

All meta-models described so far are, from the perspective of model-driven software development, platform independent models (PIM). The application models in contrast are platform specific models (PSM); they depend on the selected target platform and describe how the REST application modeled so far can be realized for a specific platform. As there is a huge amount of target platforms available, different programming languages and for each of them possibly many frameworks, our approach in general supports multiple application models. The definition and selection of the appropriate application model can, for example, depend on concrete use cases or non-functional requirements.

The application model is finally used to generate the application code, which can then be deployed and run on an appropriate target platform. Although model-driven techniques aim for a fully automated generation of application code, in general there are parts of the application where developers have to manually add specific application logic. The transformation from application model to code mainly influences how a manual adaptation of the generated code has to be performed, if it is possible to regenerate the application code without losing the manual adaptations and the general maintainability of the application. As we focus on the general approach, the set of meta-models and their relations, we will not further discuss this transformation.

F. Associated Role Model

The set of meta-models introduced and discussed before implies a role model we will introduce in the following. We propose three roles, the Domain Expert, the REST Expert and the Application Expert. Each of these roles is responsible for modeling certain parts of a REST application. Assigning each meta-model to a certain role supports to observe the principle of *separation of concerns*.

The *Domain Expert* is responsible for the definition of the domain model. As we do not prescribe any specific meta-model for that, the domain expert has to be a specific expert for the domain meta-model selected for a given use case. The role of the domain expert is independent of REST and is, as the domain model is an optional model, also an optional role involved in the design and realization of a REST application.

The *REST Expert* is responsible for the definition of the, atomic or composite, resource model as well as for the URI model. This role has to be familiar with the concept of resource orientation; it has to deeply understand the uniform interface, e.g. the semantics of the HTTP methods and their proper use. There are two different modeling scenarios a REST expert may be facing. In the *first scenario* we assume a given domain model which has been transformed into a resource model. Although we aim at automatically generating the resource model, there might be the need for manual adaptation, extension or refinement of the generated resource model. This can be caused by an incomplete transformation or by a domain model not covering every aspect of an application. In this scenario the task of the REST expert demands that he has at least a basic understanding of the domain model. In the *second scenario* we assume that there is no domain model available. Nevertheless, there has to be any kind of application description or a set of requirements provided to the REST expert. This scenario implies that the REST expert has to be able to understand the provided description or requirements as he is responsible to transfer it to an appropriate resource model. Besides the definition of the resource model the REST expert is also responsible for the definition of the URI model. This model is deeply coupled to the resource model so these two models are assigned to the same role.

Finally, the *Application Expert* is responsible for the definition of the application model and for the code generation. This role has to be familiar with the selected target platform; it has to know how to realize a prescribed REST application best suited for a platform and has in addition to consider any given non-functional requirements. Similar to the REST expert who has to understand at least the basics of the domain model, the application expert needs a basic understanding of the concept of resource oriented applications.

IV. DISCUSSION

In section II we have discussed the REST constraints and how they are fulfilled by the architecture of the WWW. We identified the main responsibilities and challenges when designing and realizing a service compliant with the REST constraints. In section III we introduced our approach for the model-driven development of REST services. In the following we will discuss how this approach, comprising a set of meta-models and transformations between them, helps to design and realize services that comply with the REST constraints.

The main responsibilities of a REST service developer are, as discussed in section II, the proper use of the uniform interface defined by HTTP, the realization of stateless applications and the observance of the HATEOAS constraint. Our approach focuses on two of these three aspects, namely the proper use of the uniform interface and the observance of the HATEOAS constraint.

The proper use of the *uniform interface*, the verbs and their associated semantics as defined by HTTP, is supported by two different aspects of our approach. At first, the use of a REST independent domain model together with an automated transformation to the resource model automatically generates a resource model that uses the uniform interface correctly. We reduce potential errors in using the uniform interface by allowing domain experts to work with a domain specific model they are familiar with. As they are not responsible to define an appropriate resource model (this is generated by the automated transformation), they cannot create an erroneous resource model. A second aspect also supporting the proper use of the uniform interface is the composite resource model. Modeling a resource structure using composite resources avoids the manual repetition of often needed modeling tasks which may introduce errors to the resource model. The composite resource model provides an additional abstraction layer simplifying the modeling of complex resource structures. The automated mapping from composite resources to atomic resources assures that at least for this part of the resource model, the uniform interface constraint is always fulfilled.

In our approach, the HATEOAS constraint is directly supported by the resource model. The resource model inherently requires to explicitly modeling the relations between the resources. In addition, the resource model is independent of any URI structure; this is defined in a separate model and does not influence the interface of the service. The URI model is only used for the implementation of the application; it has no dependency to the service description provided to clients. The automated transformation from the resource model to the application model assures that resource dependencies are mapped to links.

Besides these REST specific aspects, the use of model-driven techniques provides additional benefits like fewer repetitive implementation tasks, fewer errors associated with this, better maintainability of application code, portability through model reuse and comprehensive and consistent application documentation. On the other side, adopting a model-driven approach for the design and realization of REST services requires to thoroughly design the models and transformations and to continuously adapt them based on deficiencies that may be identified while using them [5]. Additional challenges related to model-driven software development in general are, for example, the customization and tailoring of the generated code, model evolution and versioning, as well as the complexity of managing and maintaining models as well as transformations.

V. REALIZATION

We implemented our approach as a graphical tool based on the Eclipse IDE[3] allowing modeling the models defined in section III and also implementing the model to model transformations as well as the code generation. An overview of the tool is given in Fig. 5. The domain model (1) as well as the

[3] http://eclipse.org/

resource model (2) can be defined using a graphical editor. Other models, like the URI model (3), can be defined using a simple tree editor. The transformations between the models, as shown in Fig. 1, can be parameterized. When, for example, transforming the domain model into a resource model, the user can select different strategies for this transformation. In our tool we realized this with Wizards guiding the user through the parameterization process. The decisions taken by the user are then stored in intermediate models. As shown in the upper right part of Fig. 5 (4), this leads to a set of six models involved in the process of defining a REST service and generating a service description as well as an implementation (namely the domain model, the resource model, the URL model, the domain-resource transformation model, the resource-documentation transformation model, and the resource-code transformation model).

Our tool supports a domain model comprising entities and relations as described in section III. For the service description, the generation of a HTML based documentation is supported. As application model we decided to generate a service implementation based on the JAX-RS reference implementation Jersey[4]. To increase usability, we integrated the generation of the service description and of the application code into the Eclipse IDE. As shown in Fig. 5 (4), service description and service implementation are generated as *Static Web Project* respectively as *Dynamic Web Project*. This allows for as seamless user experience in the whole process of designing and realizing a REST service.

The implementation of the models and the corresponding transformations is based on the *Eclipse Epsilon* project[5]. The models are defined in *Emfatic*, a language for defining EMF models. The model to model transformations are defined using the *Epsilon Transformation Language* (ETL) and the graphical model editors are generated using the *Eclipse Graphical Modeling Framework* (GMF). This allows us to conveniently model the meta-models and then automatically generate the code for the graphical editors. The model to code transformation is based on *Java Emitter Templates* (JET)[6], used to define templates for the generated java classes.

VI. Related Work

The approach presented in this paper is mainly based on a set of meta-models. There already exist several meta-models related to the resources model. On the one hand, there are common used REST service description languages, like for example WADL or Swagger. In contrast to our approach, these languages typically do not separate the resource model from the URI model and therefore allow violating the HATEOAS constraint. Nevertheless, it is possible to use the models defined in our work to generate service descriptions in languages like WADL or Swagger.

From a research perspective, there has also been some interesting work regarding resource models. In [19] there has been proposed a detailed meta-model for REST services. This meta-model comprises structural aspects, resources and their relations, as well as behavioral aspects, i.e. possible interactions with resources. The resource meta-model used in our work is currently a subset the one proposed in [19]. In this paper we focus on the general approach, the whole set of meta-models, their relations and the relation to REST compliance. In future work we might extend our resource model considering existing meta-models to obtain a more comprehensive resource meta-model.

The idea of using model-driven techniques in the domain of REST services is not new. In [20] a design process based on models, intermediate models and transformations is presented. This approach comprises a slightly different set of meta-models. The domain model is represented by a functional specification, mapped to a canonical information model, which

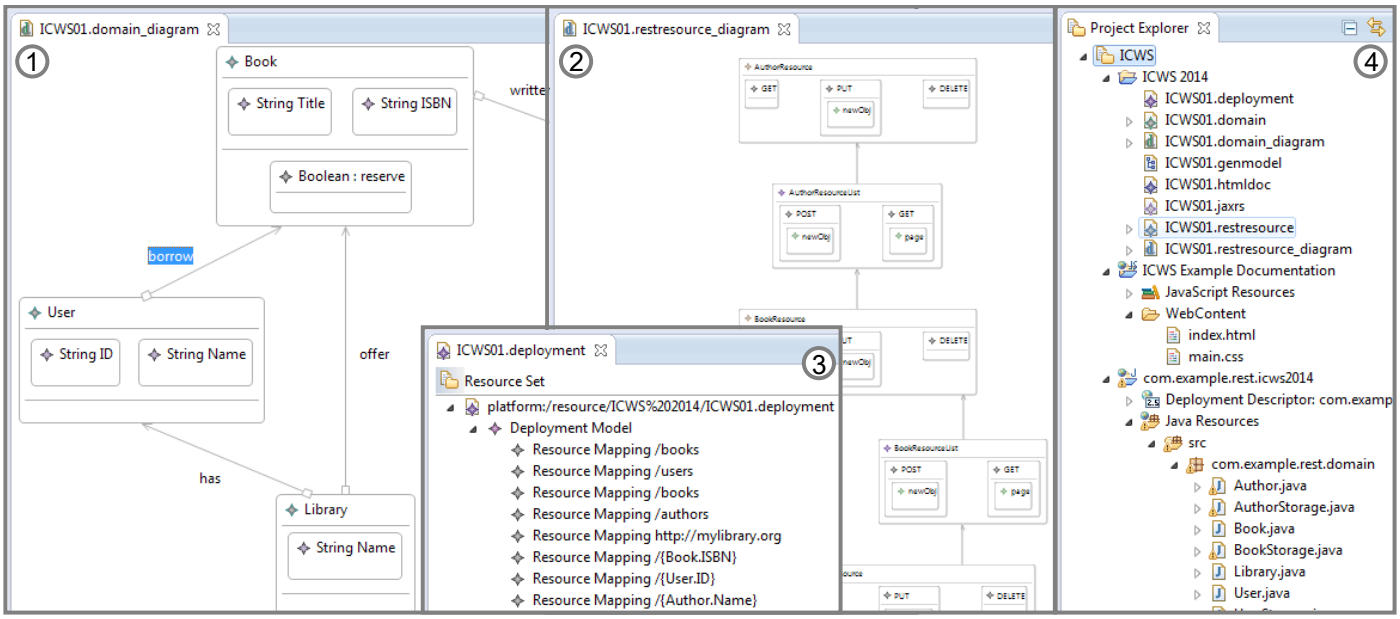

Fig. 5. Graphical tool

[4] https://jersey.java.net/

[5] https://www.eclipse.org/epsilon/

[6] http://www.eclipse.org/modeling/m2t/?project=jet

is then transformed to a resource model. The resource model presented in [20] does describe the resource structure. The interaction capabilities of resources as well as the provided and consumed representations and the URI structure are defined in a separate service specification. However, there is no separation between an abstract resource model providing a description of a service and a realization focused model comprising additional details like URI structures.

Besides the set of meta-models our approach also describes transformations between these models. In [21] it is argued that the development of correct and comprehensive model transformation is at least complicated and sometimes even impossible. Therefore, an iterative approach for the definition of model transformations is proposed. Transformations are assumed to be incomplete and imprecise; they are iteratively completed and improved. This process is demonstrated and validated using the design of a REST service as example. We agree that the definition of appropriate transformations is a non-trivial task. The transformations our approach is based on are neither complete nor flawless. They are, however, appropriate to show the feasibility of our approach and may be improved and extended in future work.

VII. Conclusion and Future Work

The design and realization of REST compliant services is still a challenging task. Concerning the WWW as target platform, the main challenges for REST service designers and developers seem to be the adaptation of the paradigm of resource orientation, the proper use of the uniform interface and the consideration of the HATEOAS constraint.

To improve this situation, we propose to use model-driven techniques and to specifically adapt them to the needs of REST. For that, we introduced a set of meta-models, discussed why we choose exactly this set and also proposed a corresponding role model. Concerning the aspect of REST compliance, the separation between the resource model and the URI model helps to fulfill the HATEOAS constraint. We keep URI information away from the resource model, i.e. the interface description that is offered to clients, and solely use it for implementation specific purposes. The usage of a domain model as well as a composite resource model allows generating the whole or at least parts of a resource model and therefore supports the observance of the uniform interface constraint. Finally we presented a graphical modeling tool that allows modeling based on the introduced set of meta-models and that also implements the transformations between these models. Our tool allows to graphically design a REST service and to automatically generate a HTML based service description as well as a service implementation based on Java and JAX-RS.

For future work, we plan to extend the single meta-models. For example, the resource model currently focuses on the static resource structure. However, the dynamic interaction with one resource as well as the dynamic relationships between resources is as well an important part of a holistic REST service model. Another aspect we are very interested in is a systematic evaluation of our approach and realization. To evaluate our work and to syndicate it with the objective declared in the beginning, we will have compare present development approaches for REST services and our approach with respect to REST compliance. This demands, inter alia, an appropriate method to capture the degree of REST compliance of a service design and realization in a comparable manner.

Acknowledgment

This work was partially funded by the BMWi project Migrate! (01ME11055).

References

[1] R.T. Fielding and R.N. Taylor, “Principled design of the modern Web architecture”, ACM Trans. Internet Technol. 2, May 2002.

[2] D. Renzel, P. Schlebusch, and R. Klamma, “Today’s top ‘RESTful’ services and why they are not RESTful”, Web Information Systems Engineering - WISE 2012. Springer Berlin Heidelberg, 2012.

[3] M. Maleshkova, C. Pedrinaci, and J. Domingue, “Investigating web apis on the world wide web”, IEEE 8th European Conference on Web Services (ECOWS), 2010.

[4] P. Adamczyk, P. H. Smith, R. E. Johnson, and M. Hafiz, "REST and Web services: In theory and in practice", REST: from Research to Practice, Springer New York, 2011.

[5] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, "Model-driven software development: technology, engineering, management", John Wiley & Sons, 2013.

[6] R.T. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–HTTP/1.1.", RFC 2616, 1999, http://www.ietf.org/rfc/rfc2616.txt.

[7] R. Burke and R. Monson-Haefel, “Enterprise JavaBeans 3.0. Vol. 5.”, O'Reilly, 2006.

[8] G. Hohpe and B. Woolf, “Enterprise integration patterns: Designing, building, and deploying messaging solutions”, Addison-Wesley Professional, 2004.

[9] L. Masinter, T. Berners-Lee, and R.T. Fielding, “Uniform resource identifier (URI): Generic syntax”, RFC 3986, http://www.ietf.org/rfc/rfc3986.txt.

[10] N. Freed and N. Borenstein, “Multipurpose internet mail extensions (MIME) part two: Media types”, RFC 2046, http://www.ietf.org/rfc/rfc2046.txt.

[11] R.T. Fielding, “REST APIs must be hypertext-driven”, http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

[12] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson, "Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more", Prentice Hall PTR, 2005.

[13] C. Ireland, D. Bowers, M.Newton, and K. Waugh, "A classification of object-relational impedance mismatch", Advances in Databases, Knowledge, and Data Applications, 2009, DBKDA'09, IEEE.

[14] L. Richardson and S. Ruby, “RESTful web services”, O'Reilly, 2008.

[15] S. Tilkov, “REST und HTTP, Einsatz der Architektur des Web für Integrationsszenarien”, dpunkt.verlag, 2009.

[16] M. J. Hadley, “Web application description language (WADL)”, 2006.

[17] “Swagger”, https://developers.helloreverb.com/swagger.

[18] R. Verborgh, T. Steiner, D. Van Deursen, J. De Roo, R. Van de Walle, and J. G. Vallés, "Capturing the functionality of Web services with functional descriptions", Multimedia tools and applications, 64(2), 2013.

[19] S. Schreier, “Modeling RESTful applications”, Proceedings of the Second International Workshop on RESTful Design, 2011.

[20] M. Laitkorpi, P. Selonen, and T. Systa, “Towards a model-driven process for designing restful web services”, IEEE International Conference on Web Services, 2009 (ICWS 2009).

[21] M. Siikarla, M. Laitkorpi, P. Selonen, and T. Systä, "Transformations have to be developed ReST assured", Theory and Practice of Model Transformations , Springer Berlin Heidelberg, 2008.

All links were last followed on 14.04.2014.