Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 50

Description Languages for REST APIs - State of the Art, Comparison, and Transformation

Anton Scherer

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. h.c. Frank Leymann
Supervisor:	Dipl.-Inf. Florian Haupt Dipl.-Inf. Karolina Vukojevic-Haupt
Commenced:	July 21, 2015
Completed:	January 20, 2016
CR-Classification:	D.2.2, D.2.11

Abstract

In recent years, the architectural style for building Web Services called "Representational State Transfer" (REST) gained a lot of popularity in industry and academia. Since designing complex, distributed hypermedia systems still meeting all the REST constraints is a difficult task, an academic, model-driven approach based on a multi-layered metamodel was developed in order to enforce REST compliance. Apart from that, multiple REST API description languages emerged in industry, providing means to formally define the structure of an API for human (e.g. API documentation) and machine (e.g. automated creation of client/server stubs) consumption. This work aims to compare the academic metamodel with API description languages widely used in industry. As a comparison methodology, bidirectional model transformations were designed and implemented between the academic metamodel and each of the two leading API description languages, Swagger and RAML. The model transformations were evaluated with a quantitative method by applying them on real world API descriptions as well as manually evaluating the quality of the transformed models. The model transformations show that indeed various mappings can be established between model elements of different metamodels. However, there are also crucial differences which are also examined in this thesis.

Kurzfassung

In den letzten Jahren erlangte der Architekturstil namens "Representational State Transfer" (REST) zur Entwicklung von Web Services eine hohe Popularität in Industrie und Forschung. Aufgrund der Tatsache, dass der Entwurf von komplexen, verteilten Hypermedia Systemen, die dennoch alle Bedingungen von REST genügen, eine schwierige Aufgabe ist, wurde ein forschungsorientierter, modellgetriebener, auf mehrschichtigen Metamodellen basierender Ansatz entwickelt, um die Einhaltung der REST Vorgaben sicherzustellen. Unabhängig davon entstanden in der Industrie mehrere Beschreibungssprachen für REST APIs, wodurch deren Struktur formal beschrieben werden kann, sodass sie von Mensch (z.B. für die API Dokumentation) und Maschine (z.B. zur automatischen Generierung von Client/Server Code-Skeletten) genutzt werden kann. Die vorliegende Arbeit vergleicht das forschungsorientierte Metamodell mit API Beschreibungssprachen, die in der Industrie häufig verwendet werden. Als Vergleichsmethode wurden bidirektionale Modelltransformationen zwischen dem Forschungsmodell und je eine der beiden führenden API Beschreibungssprachen Swagger und RAML entworfen und implementiert. Die Transformationen wurden sowohl quantitativ evaluiert, indem sie auf eine Vielzahl von existierenden, realen API Beschreibungsdokumenten angewandt wurden, als auch qualitativ durch eine manuelle Evaluation der Qualität der generierten Beschreibungen. Die Modelltransformationen zeigen, dass in der Tat einige semantische Abbildungen von Modellelementen unterschiedlicher Metamodelle hergestellt werden können. Allerdings gibt es auch wesentliche Unterschiede, die ebenso in dieser Thesis untersucht werden.

Contents

List of Figures

List of Acronyms

ATL ATLAS Transformation Language

CIM Computation Independent Model

DL Description Language

EBNF Extended Backus–Naur Form

EMF Eclipse Modeling Framework

EOL Epsilon Object Language

ETL Epsilon Transformation Language

HATEOAS Hypertext As The Engine Of Application State

HTTP Hypertext Transfer Protocol

IDL Interface Description Language

M2C model-to-code

M2M model-to-model

MDA Model Driven Architecture

MDSD Model Driven Software Development

MOF Meta Object Facility

OASIS Organization for the Advancement of Structured Information Standards

OMG Object Management Group

PIM Platform Independent Model

PSM Platform Specific Model

QoS Quality of Service

QVT Query/View/Transformation

REST Representational State Transfer

SOA Service Oriented Architecture

SOC Service Oriented Computing

SRM Swagger-Resource-Mapping

UDDI Universal Description, Discovery, and Integration

UML Unified Modeling Language

W3C World Wide Web Consortium

WSDL Web Services Description Language

XMI XML Metadata Interchange

1 Introduction

The world of Web services experienced a dramatic shift from conventional SOAP Web services to Representational State Transfer (REST). Main market drivers such as Google, Amazon or Facebook deprecated the use of SOAP Web services or even replaced them completely with RESTful Web APIs [Rod15]. REST gained widespread acceptance and the number of RESTful systems on the Web constantly grows. However, some research efforts such as [RSK12] or [ASJH11] revealed that many of today's RESTful Web services are, in fact, violating against constraints prescribed by this architectural style. This could result in loosing important benefits provided by REST such as scalability, evolvability or reliability.

In order to prevent violating against REST constraints during the design phase, Haupt et al. [HKLS14] developed an academic model-driven approach using various metamodels on different layers. This approach allows modeling and formally describing RESTful APIs. The metamodels were built considering the theoretical aspects of REST with the aim of enforcing REST compliance when designing new REST APIs.

However, there are already existing approaches widely used in industry called REST API Description Languages (DLs) which also provide ways to formally describe and design REST APIs. In addition, some vendors offer a rich tool set for extended features.

The research goal of this work is to find out if, or to what extent, the academic metamodels by Haupt et al. are compatible to the metamodels of popular REST API DLs. Semantically common and differentiating model elements must be identified. Weaknesses and strengths of these metamodels should be pointed out and finally, enhancements to the initial metamodels by Haupt et al. should be discussed.

For this purpose, the research methodology called *Model Comparison* was applied which is a fundamental discipline of *Model Transformation Testing* [LZG04]. The aim of model comparison techniques is to discover commonalities and differences between source and target models. However, instead of using automated model comparison techniques as proposed in [SC13], quasi-bidirectional model transformations are implemented manually in a transformation language called Epsilon Transformation Language (ETL). In contrast to just formally comparing the models, the model transformations provide the additional benefit to interchange API descriptions among different description formats. Comparing source and target models is a prerequisite to implement the transformations, anyway.

The following steps, which are based on the systematic process of model transformation development proposed in [SRC14], represent the procedure throughout this thesis.

1. **Finding popular REST API Description Languages** First of all, it must be identified which REST API DLs are indeed popular and widely used in industry. The results of this small market analysis yield the two candidates for the model transformations.
2. **Acquiring Knowledge about Semantic Correspondences** Both metamodels under inspection are analyzed in detail. Model elements for similar purposes in different metamodels are identified.
3. **Creating Mappings between Metamodels** This step includes establishing a conceptual mapping between the metamodels under inspection. Mapping dependencies and restrictions are identified.
4. **Defining the Transformation Architecture** Before the transformations can be implemented, the framework for running them must be established. This task requires to consider practical aspects, for example, ensuring that all metamodels are in a format that is expected by the transformation engine.
5. **Implementing the Transformation using a Transformation Language** As mentioned before, the language used for the transformations in this work is ETL which does not support bidirectional transformations natively. As two REST API DLs are compared to the metamodels by Haupt et al., four unidirectional transformations will be implemented. Although they are not pure bidirectional transformations, the effect is imitated which is why they are called *bidirectional* in a broader sense within this thesis.
6. **Evaluation** Finally, the last step is the evaluation of the model transformations. For this purpose, quantitative and qualitative evaluation methods are applied on the transformation results.

In order to provide a basic knowledge of the topics covered by this thesis, Chapter 2 gives more information about service orientation and Web services before the concepts of interface descriptions and model driven software development are discussed. Then, the already mentioned model-driven approach by Haupt et al. is presented with a specific focus of the part which will be relevant for the model transformations. Afterwards, the process of finding and analyzing REST API DLs used in industry is the significant part in Chapter 4. The main work of this thesis, Chapter 5, covers the conceptual design of the model transformations. Furthermore, a third conceptual mapping, which will not be implemented, is presented in this Chapter, involving a relatively new REST DL called RADL. Afterwards, some implementation-related issues are explained before Chapter 7 evaluates the implemented model transformations. Chapter 8 discusses the benefits of each approach before proposing some improvements to the existing metamodel by Haupt et al. based on the new findings. Finally, a conclusion summarizes the most important aspects of this work.

2 Background

This chapter reveals the fundamental knowledge needed for comprehension of the following sections. After introducing Web services, their context and types, some basic information about interface description languages, especially for REST and SOAP, will be discussed. Finally, the last section discusses the idea and most important aspects of Model Driven Software Development (MDSD).

2.1 Service Oriented Architectures and Web Services

Heterogeneity is the common case in today's software systems. Different platforms, programming languages, data formats or protocols make integration and collaboration of applications very hard. A common programming interface and an interoperability protocol would reduce the complexity and existing functionality could be reused [BS11]. In fact, this is one promise of Service Oriented Computing (SOC), a paradigm using services to support distributed, evolvable and interoperable applications. SOC's vision is to assemble a set of services performing dynamic business processes spanning over mulitple organizations. The key to realize SOC is Service Oriented Architecture (SOA) [PTDL07].

According to the Organization for the Advancement of Structured Information Standards (OASIS), which is a non-profit consortium for adoption of standards, "*SOA is a paradigm*[1] *for organizing and utilizing distributed capabilities that may be under the control of different ownership domains*"[OAS06]. These capabilities are offered as services which are loosely coupled, coarse-grained and realize autonomous business functions. Each service exposes behavior through contracts. Services can be reached through discoverable network addresses, called *endpoints*, via various transport protocols and formats, supporting different non functional aspects. A service is *always on* meaning that it does not have to be created by clients before calling it [RGO12].

The roles of SOA can be pictured as a triangle, called *SOA Triangle* [WCL+05] as illustrated in Figure 2.1. First of all, the service provider publishes his service description about the service he wants to offer to the service directory. One typical example for a service description language for SOAP Web services is Web Services Description Language (WSDL) (see Chapter

[1] more precisely, it follows the SOC paradigm

2.2.1). Afterwards, service consumers are able to look up services in the service registry. They can browse available services and finally find details required for interacting with them. Universal Description, Discovery, and Integration (UDDI) is a standard XML-based registry offering publishing and finding capabilities. Finally, service consumers, who can also be service providers to other clients, establish a connection to the corresponding service provider called *binding*. However, the service discovery does not mediate between provider and consumer since it is independent of the binding process. Yet, the registry plays an important role as it decouples the other two roles: Provider and client do not need to know about each other in advance. Although the SOA Triangle is independent of the underlying technology, the SOAP message format (see Chapter 2.1.1) is the predominant choice for realizing it.

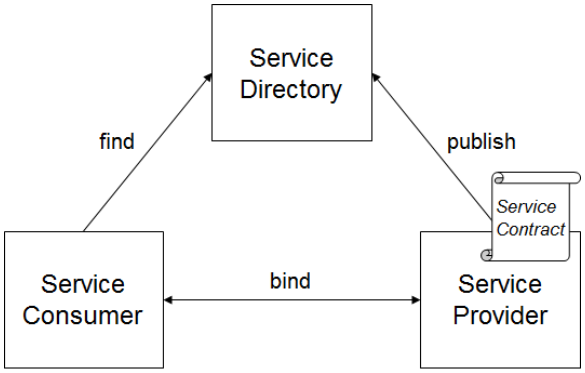

Figure 2.1: The general SOA Triangle

Web services realize the SOA concept. The common term *Web service* might be confusing since it is used in many different ways. In fact, there is no universal or official definition but rather multiple suggestions highlighting various aspects. For instance, the Web Services Architecture Working Group [OMN+04] of the World Wide Web Consortium (W3C) defined a Web service as a "*software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*" This definition highlights the aspects of interoperability and network distribution, but it can be criticized that it is too technology-specific as it is tailored to SOAP Web services.

In contrast, the following definition by Papazoglou [Pap08] encompasses a broader context, highlighting the autonomy of a Web service : " *A Web service is a self-describing, self-contained software module available via network, such as the Internet, which completes [business] tasks [...]*".

Instead of providing further definitions, the following list (adopted from [Ecl07]) summarizes important characteristics of Web services.

self-descriptiveness The service provider publishes a description of its interface providing the client with sufficient information on how to interact with the Web service. Thereafter, format and content of request and response messages are settled.

self-containment Web services have explicit boundaries. They should not depend on other services to execute.

composable and reusable Simple Web services can be aggregated to more complex ones. These can be executed by workflow engines.

platform- OS- and language-independence Interoperability between different technologies is ensured by standardized interfaces and message formats.

machine-to-machine communication Web services do not need internet browsers, HTML or other GUI technology as there is no human intervention in the communication process, typically.

discoverability Web services should be supplemented with meaningful metadata by which they can be effectively discovered.

It should be noted that some of the mentioned aspects are also applicable for the higher-level, non-standardized SOA principles [Erl07]. The following sections describe the most important two types of today's Web services: The SOAP and REST style.

2.1.1 WSDL-based Web Services

WSDL is an XML format providing an extensible model for describing Web service interfaces and how to access them. Initially, WSDL was developed by Microsoft and IBM [New02]. Basically, a WSDL machine-readable file describes *what* services are available, *where* they are located and *how* they can be invoked.

The most recent WSDL specification is version 2.0. However, at the time of writing, version 2.0 has not gained wide adoption in industry. Moreover, most tooling support is based on version 1.1 [BPR13]. Anyway, the new WSDL version has not changed the basic concept since only minor corrections are made, for instance renaming some terms (e.g. `<portType>` renamed to `<interface>`, `<port>` renamed to `<endpoint>`) or removing `<message>` definitions which are now defined in the `<types>` element [BPR13]. Figure 2.2 shows the structure of a WSDL description in version 1.1.

The root of every WSDL file is the `<definitions>` element encapsulating the entire document and providing several namespace definitions. As discussed before, a separation between abstract and concrete descriptions can be made. The first abstract description element is `<types>` where the data types of exchanged messages between client and server can be defined. As suggested before, XML Schema should be the preferred choice.

Figure 2.2: The structure of a WSDL (version 1.1) document

The next element is `<message>` providing an abstract information about input, output or error of an operation in form of one or more logical parts. The last abstract element is `<portType>` defining *what* the Web service does. It describes the executable operations and its corresponding input and output messages.

The first concrete element to be discussed is `<binding>`. It binds `<portType>` definitions to a data format specification and a concrete transport protocol. Furthermore, it contains an `<operation>` element for each operation in the `<portType>` it is describing. The bindings can be offered via multiple protocols, for instance over Hypertext Transfer Protocol (HTTP) (with GET or POST method) or SOAP. Finally, the `<service>` element defines where to access the service. It contains one or more supported ports, each port representing an access point. A port specifies a unique address and it is associated with its binding.

Actually, the combination of WSDL with the already mentioned SOAP protocol is a very popular way of building Web services in practice. But it should be noted that WSDL is protocol independent (see Chapter 2.2.1). Originally, SOAP was an acronym for *Simple Object Access Protocol*. Since SOAP Specification Version 1.2, the W3C dropped this expansion due to its misleading nature: SOAP accesses procedures or functions which do not have to support object-oriented programming concepts such as encapsulation, object instantiation, etc. Therefore, the W3C redefined the term SOAP without even mentioning the term "object":

"*SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. SOAP uses XML technologies to define an extensible messaging framework, which provides a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics.*" [MMG+07]

According to that quote, the key points of SOAP are extensibility and independence of protocol, programming language and platform. SOAP is a message architecture on top of a transport protocol such as HTTP in order to define the structure of XML messages as illustrated in Figure 2.3. The envelope is the root element indicating the beginning and the ending of the SOAP message. The header element is optional, containing any number of child header blocks. Each block might define routing information or Quality of Service (QoS) configurations such as security, transactions or reliability. Thanks to SOAP's extensibility, additional headers can be introduced, supporting new features. Finally, the SOAP body is a mandatory container carrying the payload of the message for the ultimate recipient.

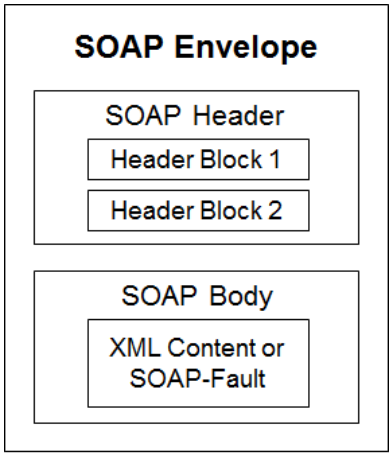

Figure 2.3: The general structure of a SOAP message

Furthermore, SOAP defines a distributed processing model with three important node types: The *initial sender* where the message originates, the *ultimate receiver* representing the final destination and *intermediaries* which are nodes along the message path. According to their role, intermediaries might process specific header information. For example, they could add or remove data, produce a fault message or just forward the original message to the next hop. Therefore, intermediaries are sender and receiver of SOAP messages [MMG+07].

Besides the mentioned standardized message format and the processing model, the SOAP specification also includes mapping conventions for mapping application data into SOAP messages and network protocol bindings for HTTP, SMTP and others.

Yet, one crucial strength of SOAP is its protocol transparency and independence: A SOAP message can be transported over various transport protocols which may change along the message path. Also QoS aspects can be preserved since their definition is independent of the underlying protocol. Both features are made possible through SOAP header definitions [PZL08].

On the other hand, performance issues could arise due to the (un-)marshalling process and because of the verbose XML format. Interoperability problems are possible if native data types or language constructs are described in WSDL. In addition, the extensive and complex WS-* stack of standards might be hard to learn for inexperienced developers.

2.1.2 REST-based Web Services

In 2000, Roy Fielding published his doctoral dissertation [Fie00] and introduced the term REST as an architectural style with a set of constraints for distributed systems on the Web. Since these constraints are hard to check in a formal way for real world services, there are no hard rules or a set of standards to follow.

"*REST is an architectural style that, when followed, allows components to carry out their functions in a way that maximizes the most important architectural properties of a multi-organizational, network-based information system. In particular, it maximizes the growth of identified information within that system, which increases the utility of the system as a whole.*" [Fie08a]

While this statement is quite abstract, the following section describes the mentioned "important architectural properties" and how they can be optimized.

REST Key Constraints

Fielding defined five[2] mandatory constraints in his work which will be discussed in the following.

1. **Client-Server** One of the most fundamental constraints is the architectural separation of a service requester and service provider. Clients are responsible for the user interface and the processing of responses, while servers handle the storage capabilities and server-side processing. Clearly, this principle of *separation of concerns* is followed by the Web naturally when considering Web browsers and Web servers.

   Consequently, the distributed architecture provides the benefit of components evolving independently from each other as they only rely on common uniform interfaces (see *4. Uniform Interface*). Moreover, clients could run in heterogeneous environments, improving their portability. Finally, the server-logic, which becomes simpler because it does not contain front-end logic, is more scalable.

[2] the sixth constraint, called *Code on Demand*, is optional and will not be discussed

2. **Statelessness** Client-server interactions have to be stateless, meaning that all information required for processing a client's request, must be contained in it. The client must not expect the server to use any stored context or session information. Thus, the session state must be kept on the client. Essentially, this constraint improves three architectural aspects: (1) Visibility since monitoring is more simple as all information is contained within the request, (2) scalability because servers do not have to store state as they don't have to manage resource usage across requests and finally (3) reliability which is improved by quickly recovering from server failures.

   On the other side, the trade-off is an increased network overhead since session data must be transferred repeatedly. In addition, clients become more complex because of maintaining the session state.

3. **Cache** The cache constraint requires that a server response must indicate somehow whether the response is cacheable or not. If so, the response data can be reused for later, equivalent requests. In general, caches can be realized in three different layers [RAR13]: Typically, the Web browser (1) manages the cache on the client side. Subsequent requests whose response was cached do not cause any network traffic. Moreover, caches can be realized on intermediate nodes (2), for instance on proxies. Thus, the request to the actual backend can be saved, reducing its workload and the request's network latency. Alternatively, the cache can be implemented on the destination server (3) by storing the responses of frequently incoming requests in memory which can be retrieved in a fast manner.

   The downside of caches are that they decrease reliability since the actual resource could differ significantly from the representation of the cache. This could happen if the cache is not updated while there has been major changes to the resource. However, HTTP provides a set of features making it convenient to work with caches. For example by using the header field *cache-control*, by setting an expiry date or by using *etags*.

4. **Uniform Interface** According to Fielding, the uniform interface is the central feature of REST differentiating it from other approaches. It is a general, overarching way of designing the technical interface between service consumer and service producer. Resources are accessed through four basic operation types: create, read, update and delete (CRUD).

   In order to support a proper building of uniform interfaces, the following four additional constraints are defined.

   Identification of Resources Resources are the building blocks of a RESTful API as they are offered to be interacted with. They are addressable by a *Uniform Resource Identifier (URI)* which is uniquely defined in a global context. While a resource's state may change, a URI typically does not.

Manipulation of Resources through Representations Resources could be any kind of entity, for example, physical objects or abstract concepts which cannot be sent over network. Therefore, the concept of representations of resources as a "sequence of bytes" was introduced. Representations are structured data that can be transferred. A resource can be represented in different formats, for example in XML or JSON. Usually, client and server negotiate about the representation to be fetched which is called *content negotiation*. HTTP supports typed payloads by using media types which are precise and openly available specifications of the semantics of the representational data format.

Self-descriptive Messages Each message should contain all information required to understand it completely so that no out-of-band information is required. If comprehending the message content does require some meta-data, they should be referenced in a header field, for example the media type definition in the *Content-Type* field. The stateless constraint is just a special case of this requirement as it demands requests to be autonomous [RAR13].

Hypertext As The Engine Of Application State (HATEOAS) Hypermedia is the driving force for changing the application's state. The only way for a client to change its state is to fire HTTP requests and processing the response. The key question is: How does the client know which actions can be performed in the current state? In order to answer this question, representations need to include hypermedia links which can be discovered dynamically. This can be compared to HTML providing hyperlinks and forms which can be used to navigate to related resources. Ideally, the client needs no prior knowledge to interact with a RESTful service except a root URL.

The evolving system can be seen as a directed graph: The vertices represent the resources while the edges connect the resources through hypermedia links. URI links are used to navigate from one resource to another.

Advantages of uniform interfaces are a simplified system architecture, visibility of interactions and decoupling from the service implementation, leading to the benefit of independent evolvability. One disadvantage is the missing possibility to customize interfaces for specific scenarios which would result in more efficiency.

5. Layered System In order to reduce the system's overall complexity, REST prescribes that components must be arranged into hierarchical layers. Each component does not have any knowledge about all the other layers except the immediate layer it is interacting with. By meeting this requirement, intermediaries between client and server can be installed without breaking the application. Thus, routers, caches or load balancers could be used as separate components.

> The layered system constraint improves scalability but it also could add additional overhead because of network latency and extra processing time. However, implementing caches significantly increases the overall end user performance in many cases.

REST has gained a tremendous popularity within recent years at industry and academia. Though, it is often misinterpreted by people trying to build RESTful services. However, compared to the WS-* specifications for SOAP/WSDL Web services, REST is perceived to be simpler because it builds on well-known standards such as HTTP, XML, URI or MIME types.

Besides the mentioned non-functional improvements such as scalability, evolvability, reliability or visibility, another main advantage of REST is its low barrier for adoption. There is no special tooling or infrastructure needed, clients to REST services can be build fast and the deployment of a RESTful Web service is easy. It offers the possibility of a lightweight message format such as JSON or even plain text which optimizes the performance of the service [PZL08]. Moreover, the process of reusing and combining atomic services in order to achieve sophisticated functionality, called *service composition*, can also be realized for REST architectures as pointed out in [HFK+14].

On the downside, the semantics of HTTP methods could be misused in order to use HTTP just as a tunneling protocol. For example, all requests could be send via the GET method, even when updating, creating or deleting a resource. While this is convenient for clients with limited HTTP vocabulary or for firewalls blocking some HTTP methods, it severely violates against the constraint of the proper use of uniform interfaces. Furthermore, various RESTful Web services tend to be *chatty* with frequent client-server interactions decreasing the network performance.

Although REST is protocol independent, HTTP is the predominantly used protocol. That is why this protocol will be described in the subsequent section.

HTTP

Beyond doubt, the Hypertext Transfer Protocol is the most used application protocol in the context of the World Wide Web. The Internet Engineering Task Force (IETF) proposed several standards, by publishing Requests for Comments (RFCs). According to RFC 7230 [FR14a], HTTP is based on a distributed client-server architecture using extensible semantics for flexible interaction with network-based hypertext information systems. The protocol defines two types of messages: Request messages which can have multiple intentions (see HTTP methods below) and response messages including a status code and possibly relevant information such as the requested resource, error information, etc.

Every HTTP message consists of a *start line*, header fields and an optional body. Syntactically, the start line distinguishes between the two types of messages: Request messages indicate the

HTTP method, the request-target and the HTTP version while response messages contain the HTTP version, the status code and an optional description of the status code.

Actually, header fields, containing the meta-data of the message, play a major role in the context of RESTful services. They offer standardized, dedicated header fields for important mechanisms such as caching (e.g. with the header *cache-control*) or content negotiation (e.g. with multiple *accept*-headers) which is a principle for finding the best suited representation of one specific resource for the client. By design, HTTP is a stateless protocol supporting the self-descriptiveness of exchanged messages.

The most important constraint of REST, the uniform interfaces, is realized in HTTP with its methods[3], prescribing specific method semantics. The most commonly used ones are explained in the following list.

GET This method is used to retrieve a representation of a resource. It is considered to be a *safe* method as it does not modify resources. Furthermore it is *idempotent* since the request can be made any number of times with the same outcome.

PUT Requests sent over PUT demand that the enclosed entity should be stored under the request URI. If there is an existing resource for this URI, the entity enclosed in the message should fully replace the old version. PUT is not safe but idempotent.

POST This request contains an entity to be accepted as a new subordinate of the resource. Actually, the origin server decides how to process this request. The result could be a new created resource, a change to an existing resource or initializing a data handling process. POST is not safe and not idempotent.

DELETE This method requests that the resource which is addressed by the URI should be deleted if present. DELETE is not safe but idempotent.

In addition, HTTP defines a huge number of response codes, which also have a predetermined semantics, that should indicate the client how the request has been processed. For example, 200 (OK) means that the requests has been processed as expected, 204 (Created) signifies that a new resource has been created and so on. A complete reference of response codes can be found in RFC 7231 [FR14b].

[3] sometimes also called *verbs*

2.2 Interface Description

Developing against a set of abstract interfaces is a key factor of building distributed systems. In general, an Interface Description Language (IDL)[4] is a language used to define the interfaces between consumer and service producer in a distributed system as a textual description. It serves as a contract between service consumer and service producer. In fact, the concept of IDLs is not new. In 1981, Wulf, Nestor and Lamb [NLW81] developed one of the first IDLs which constitutes as a basis for many others to come later. They already defined important concepts of IDLs which are still relevant today. The most important ones are listed in the following.

Machine Independence Data which are described in IDL can be communicated between different programs and machines. Back then, the authors developed a standard ASCII representation format of the data which can be transported over a network.

Language Independence Although client and server could be implemented in different programming languages, they are able to communicate with each other. A tool named *IDL translator* generates *readers* and *writers* (today known as client and server stubs) for mapping between concrete internal representation and abstract exchange representations.

Separation of Specification and Implementation Interfaces are described in an abstract way without the influence of implementation-specific details. This enforces the information hiding principle and the decoupling between clients and servers.

Two of the earliest description languages which were adopted by industry were the IDL developed by the Object Management Group (OMG), specifically designed for Common Object Request Broker Architecture (CORBA) systems and the IDL developed by Microsoft called MIDL, exclusively tailored to Component Object Model (DCOM) systems [SR03]. Their bridging strategy consisted of two parts: First, it must be agreed on *how* to make invocations regarding data encoding, naming, or activation principles. Usually, this was done by the distributed computing standards. On the other hand, *what* to call was specified within the IDL, e.g. operation names, signatures, or return types [GDD+04].

Due to the technology dependency, for instance the transportation formats (IIOP for CORBA and DCOM for COM) and the lack of extensibility, these IDLs were not interoperable across different target environments [Heu07]. Clearly, those IDLs are not suitable for the Web where heterogeneity is common. WSDL solved the mentioned shortcomings by providing a technology-independent and extensible description language.

[4] sometimes also referred to as *interface definition language*

2.2.1 WSDL

Since proprietary IDLs were not suitable for the Web, WSDL filled this gap as a standardized way of describing interfaces of heterogeneous systems. According to [WCL+05] WSDL underlies the following concepts:

Extensibility Additional properties which are not part of the original WSDL grammar can be attached to the service description, for instance quality attributes of the service in form of WS-Policy.

Support for multiple Type Systems Besides XML Schema, WSDL supports arbitrary other type systems, for example doctype definitions or even platform-dependent ones. However, it is recommended to use XML Schema in order to increase the likelihood of consumer interest.

Unifying Messaging and RPC Often, Web services are applied as a unifying integration architecture using RPC or messaging. WSDL supports both flavors.

Separation of "what" from "how" and "where" WSDL splits its content into two parts: *abstract* definitions containing data types, message patterns as well as abstract operations and the *concrete* part describing how the service is to be interacted with (message format and protocol information) and where the service is offered. This separation enables reuse of abstract service descriptions.

Support for multiple Protocols and Transports Although SOAP over HTTP is the most common way of interacting with WSDL-based services, also other protocols are supported. For instance, the same service might also be available over RMI/IIOP over TCP at another location.

No Ordering The description does not indicate in which order to execute multiple operations.

No Semantics WSDL only describes the functionality of the service at a structural level. The semantics have to be specified separately, for example, by using Semantic Annotations for WSDL and XML Schema (SAWSDL) which is a W3C recommendation [5].

However, describing the interface separately from its implementation exposes a maintainability problem in practice. If the implementation code changes affecting the interface, the WSDL file will get out of date. However, there is tool support generating the WSDL file automatically after the implementation code has been changed.

[5] http://www.w3.org/TR/sawsdl

2.2.2 API Description of RESTful Services

In contrast to human facing Web applications which should be self-describing to their human users, machine-consumable Web services typically do need a descriptive documentation about how to interact with them. In fact, it might be argued that the required self-descriptiveness and the dynamic discovery of resources at runtime as proposed by the HATEOAS principle would obviate the need for separated, static API descriptions [Ram14]. Furthermore, separated documentations would require additional maintenance effort in order to keep the documentation up-to-date. The following quote by Fielding reinforces this opinion:

"*A REST API should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types. Any effort spent describing what methods to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type (and, in most cases, already defined by existing media types).*" [Fie08b]

While this statement expresses what should not be documented, it clearly sets the focus on documenting custom media types which carry the semantics of the operations. However, it does not state *how* they should be described. When a client uses a truly RESTful service with no prior knowledge, the client knows nothing more than an initial URI and a set of media types. For the API provider the question remains where and how to organize the required information, including application-specific semantics [RCSW13]. As several research efforts confirmed [VSVD+12] [SA11], this is best done by an out-of-band documentation in the form of REST API descriptions. Furthermore, static descriptions provide an essential usability improvement for the clients in order to get an overview of the API's capabilities. Usually, this is referred as the Application Programming Experience (APX) which significantly improves with a formal contract that reflects the structure of the API [RAM15b]. Otherwise the API must be discovered dynamically by making lots of requests trying to get an overall picture. For some APIs, this is not an option at all, for example, if each request is subjected to a charge or the number of resources grows dynamically so that the consumer will never be sure that actually all resources have been discovered.

In contrast to WSDL for SOAP Web services, no de facto standard has evolved for describing REST Web Services by now [VB15]. However, in the last few years some REST API description languages penetrated the industry market, for example, Swagger, RAML or API Blueprint. The goal of these languages is to provide features for describing interfaces for REST APIs which can be read by humans and machines and which are not dependent on specific programming languages. By now, their popularity is rising. Currently, a consortium called Open API Initiative works on a vendor agnostic standard format for describing REST APIs which is based on the Swagger specification (see Chapter 4.2.1).

Generally, documentation can be written or generated for humans and/or for machine processing. Companies historically relied on manually written documents, typically presented as HTML pages, to expose REST contracts to clients. Often, example requests which are not intended to be machine-readable should indicate how to interact with the API. Even popular APIs such as the Twitter API do not have machine-readable descriptions but only human-focused textual documentation [VHM+14]. As the structure, format and style of human-centered documentations differ drastically, it is very cumbersome to automatically extract valuable information with machine processing.

In contrast, producing machine-readable descriptions provides several benefits. First of all, human readable documentation can be generated which is not only visually appealing but also full of smart features. For example, the tool *Swagger UI* reads a Swagger definition and generates a user interface documentation which provides testing features by enabling the user to fire example requests easily. Furthermore, client stubs could be automatically created in multiple programming languages [VHM+14]. Going one step further, it is also possible to generate automated test cases out of the API description. In addition, real-time validation of request messages can be performed as their message format is predefined. Interoperability is another important aspect as descriptions in XML or JSON can be easily retrieved across web or mobile implementations. Finally, considering that a large enterprise could provide hundreds of APIs, formal descriptions will ease the design and management of these APIs, for example due to inheritance or reuse of common schemas.

Another potential advantage in the future would be accomplishing higher level tasks such as programmatic API discovery. Though, this exposes several problems in practice, since machine-readable descriptions still could have different data formats, use heterogeneous vocabularies or lack non-functional properties which are crucial to discover services, e.g. licensing restrictions, service usage limits or QoS properties. Panziera and Paoli [PP13] address these mentioned issues by proposing a collection of best practices and a framework in order to generate self-descriptive, standardized (based on RDF) service descriptions.

Common Description Features

In fact, many popular REST API DLs support similar description features. Without going into details of specific languages, the common definitions usually include the following building blocks:

Global Information General information about the author, the API version, the base URL, etc.

Endpoints The service entry point of a resource. Usually, a path is used relative to the base URL.

HTTP Methods Available HTTP verbs are defined on a resource.

Input Parameters The definition of input parameters includes its location, e.g. in the header fields, as a query, path or body parameter. The scheme of the parameter can be specified, too.

Output Parameters Concerning the response message, output parameters usually include the response code and/or the scheme of the response message.

Media Types Incoming and outgoing media types can be specified for each operation.

Data Schemes As already mentioned, the scheme of parameters can be defined within the document or referencing to external schema definitions.

Documentary Information Some languages provide extended documentation capabilities such as descriptions for parameters, the purpose of a resource, comments, operation IDs, etc. Mainly, this documentation is necessary for generating meaningful human-readable documentation.

Most REST API DLs provide features for defining security and authentication information, others define specific constructs for error handling. However, the leading REST API DLs in industry do not support hypermedia API definitions (See Chapter 8).

2.2.3 Development Styles

When building Web services, regardless if REST- or WSDL-based services, generally one can classify the following two development styles: *Contract-last*[6] means starting directly with the implementation code. The interface description file will be generated by special tools. On the other hand, the *contract-first*[7] approach requires developers to design the interfaces first. This is done by writing the formal interface description where the implementation code can be generated from. A REST specific design approach of contract-first is proposed by Haupt et al. by modeling the Web service interfaces and generating code out of the models (See Chapter 3). Usually, the generated code must be enhanced with additional business logic, manually. Table 2.1 shows a comparison of both approaches regarding some important non-functional aspects.

Because of the mentioned benefits of contract-first, this is the preferred choice of many developers, especially when designing more complex systems exposing functionality over Web services. However, developers need a deep understanding of the description language in order to model well-designed services.

[6] also known as bottom-up
[7] also known as top-down

The tool support for contract-first and contract-last depends on the interface description language. WSDL has tool support for both approaches while RAML (see Chapter 4.2.2) clearly is a contract-first approach of designing RESTful Web services.

In fact, the development style influences the documentation of the APIs, too: If the contract-first approach is used, the documentation of the API can be part of of the design phase. This allows changing the description without affecting the production code. The downside is that the documentation can get out of synchronization with the program code easily. On the other hand, the contract-last approach enables to document the APIs within the code, generating the description afterwards. As the documentation is located right next to the code, it increases the likelihood that developers change both. Though, even a minor modification of the documentation could mean browsing and editing the production code.

Table 2.1: Comparison of contract-last vs contract-first. Based on [ML09] and [PER13]

	Contract-last	**Contract-first**
Ease of Development	Easy to use and fast to deploy a running Web service	More complex and more time consuming to design the contract.
Maintenance	Implementation code is likely to be changed. As the contract depends on the code's internal API, it might be necessary to change the service description as well. Finally, clients must be adopted to the new version.	The external API is less likely to be changed frequently.
Portability	As the contract is based on the previously written code, it could contain implementation-specific constructs which are not available in other languages. When relying on the implementation code, portability is typically limited.	Usually, the contract is written in data structure format languages such as XML, JSON or YAML which all are independent of platforms and programming languages. They can be mapped to various programming languages afterwards.
Performance	When transforming Java to the contract description, the developer has no control over the objects to be serialized. This might result in unnecessary data which is sent over the network increasing the network load.	The developer explicitly specifies which data should be sent at which operations. Typically, this results in a more efficient design.
Reusability	Usually, the definition parts of generated descriptions cannot be used in other scenarios as they highly depend on each other.	Typically, the contract-first approach results in granular components within the description file which can be reused in the same or in other descriptions.

2.3 Model Driven Software Development

When building large-scale applications, implementation on source code level might have several drawbacks, for example the system's code complexity is hard to manage and the code quality highly depends on the skills of developers. It is difficult for humans to keep an overview of the system by examining low level programming code. In contrast, Model Driven Software Development (MDSD) aims to define software functionality on a higher level by using abstract and formal models which can then be transformed to concrete and executable implementations [Rec08]. Instead of providing a way *how* to implement functionality, MDSD rather describes *what* should be implemented. In fact, generating code based on models is nothing else than a model transformation (see Chapter 2.3.3).

There are several advantages of MDSD influencing not just the quality of the software development process but also important economic factors: First of all, MDSD can increase development speed through automation of model transformations. One model element typically represents multiple lines of code. Complexity is managed through abstractions, increasing productivity and effectiveness of developers. Furthermore, they are freed of the burden of programming repetitive or standard tasks where unnecessary errors can be avoided by automation. Rather, programmers can focus on modeling actual functionality. If they want to change cross-cutting implementation concerns (e.g. logging or exception handling) they can do so by changing one section without examining the entire source code. Code quality does not depend on human programming skills but on the code generator. By using best practices and design patterns, the code generator can produce high quality code which will recur uniformly. Non-functional aspects such as performance could also be improved by the generator. Finally, domain experts can directly be involved in the development process, making their knowledge available in the form of domain models (see Chapter 2.3.2) [SVC06].

Drawbacks of MDSD are that developers could over-simplify real world issues when raising the level of abstraction. MDSD is a paradigm shift forcing developers to think on a conceptual level. Some might be overwhelmed as they do not have the required expertise. Since multiple models formalize different views on the same object, redundancy and consistency management might get cumbersome. In addition, so called *round-trip problems* could occur: Complex systems will have a lot of models and levels of abstraction which are interconnected. If one interrelated model is modified, other models on multiple levels of abstraction might need to be changed, too. However there is appropriate tooling support for solving these issues [HT06].

2.3.1 Models and Metamodels

Models are the main building blocks in MDSD. Völter and Stahl [SVC06] define models as "*abstract representation[s] of a system's structure, function or behavior*" that adhere to a specific structure. This general definition infers that models are not bound to software engineering as

they can be applied in many scientific fields. In the context of MDSD, models are a concept in order to separate the specification of functionality of a system from its technology. Based on a classical definition by Stachowiak [Sta73] models have three main properties: abstraction, isomorphism and pragmatism.

Abstraction Models can be defined on different layers, visualizing different focus areas with regard to the architectural concepts. Therefore, irrelevant details can be ignored, whereas the main aspects are represented by elements of the model. This perspective is also called a *viewpoint* on a software system [OMG03].

Isomorphism Models are a projection of real world entities. That is why a certain degree of correspondence must be established by mapping important attributes of the real world entity to model attributes. Although unconsidered attributes are not included, conclusions about the real world object should be drawn by the model.

Pragmatism Each model has a goal which the model was created for. The model purpose determines the level of abstraction and isomorphism.

Models can be created using modeling languages such as the popular Unified Modeling Language (UML). In fact, modeling languages are defined by a metamodel in order to establish a set of rules for each model instance. Happe defines the term metamodel as a "*precise definition of the constructs and rules within a certain domain needed for creating semantic models on certain level of abstraction*" [Hap14].

A metamodel formalizes relevant concepts of a domain which is a field of interest with common constructs and rules [SVC06]. Metamodels do not have to cover the complete domain but rather a specific extract. Domains can be divided into smaller subdomains, for example the GUI layout domain can be separated from its persistence layer.

Metamodels can be imagined as a building set providing all information in order to create model instances. They define an *abstract syntax* of the elements and their relations in their model instances. The *concrete syntax* defines a mapping from general language constructs to the actual representations. While there can be only one abstract syntax per metamodel, multiple concrete syntaxes are allowed, for example, models can be represented in XML or JSON format. However, the concrete syntax is not part of the metamodel [SVC06].

Moreover, metamodels provide validation rules by which the *static semantics* of the models can be validated. For instance, rules are defined how modeling primitives can be composed. Models conforming to a metamodel must be valid with respects to its validation rules. Additionally, the meaning of well-defined model constructs is described as part of the *dynamic semantics*. It defines how to interpret valid model instances. Usually, this is described in text format [Hap14].

A metamodel is a model instance of a *meta*-metamodel which, in turn, also should have a metamodel. In order to avoid an infinite loop, meta-metamodels are *self-defined*[8]: The structure of a meta-metamodel is again specified in the meta-metamodel itself. The meta-metamodel contains all universal features in order to build metamodels. Several self-defining meta-metamodels have been proposed. The Object Management Group (OMG) introduced the Meta Object Facility (MOF) which is a popular standard for defining a common specification of metamodels [OMG06]. For example, Figure 2.4 uses UML adhering to the MOF standard, in order to illustrate the four-layered MOF model. In this view, each layer is an instance of the overlying metalayer and describes the underlying one. The real world object is described by a UML class which in turn is defined by the UML metamodel, specifying the concept of UML classes. Finally, the highest layer M3 is the self-defining meta-metamodel MOF. Another self-defining meta-metamodel standard is called *Ecore* which will be introduced in the following section.

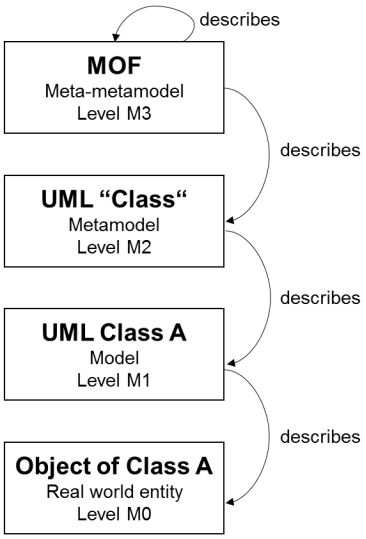

Figure 2.4: Example of the four-layered architecture of MOF

Ecore

Ecore is the metamodel of the Eclipse Modeling Framework (EMF) models. EMF is a Java-based modeling framework and code generation facility. It is used to build tools and other applications based on a structured data model. Data models can be represented for example in XML, Java and UML . The canonical data format of an Ecore model definition is XML Metadata Interchange (XMI) [Ecl15].

[8] also called *reflexive*

According to the Eclipse Help Website [Ecl05], EMF started out as an implementation of a subset of the MOF specification. Since MOF version 2.0, a similar subset of the MOF model has been separated out, called EMOF (Essential MOF). There are slight differences, mostly naming conventions, between Ecore and EMOF. Nevertheless, EMOF instances are supported by EMF.

As discussed before, Ecore is itself an EMF model, representing a meta-metamodel. Because of clarity reasons, Figure 2.5 shows only a subset of the Ecore model. Steinberg et al. [SBPM09] call this subset the *Ecore Kernel*.

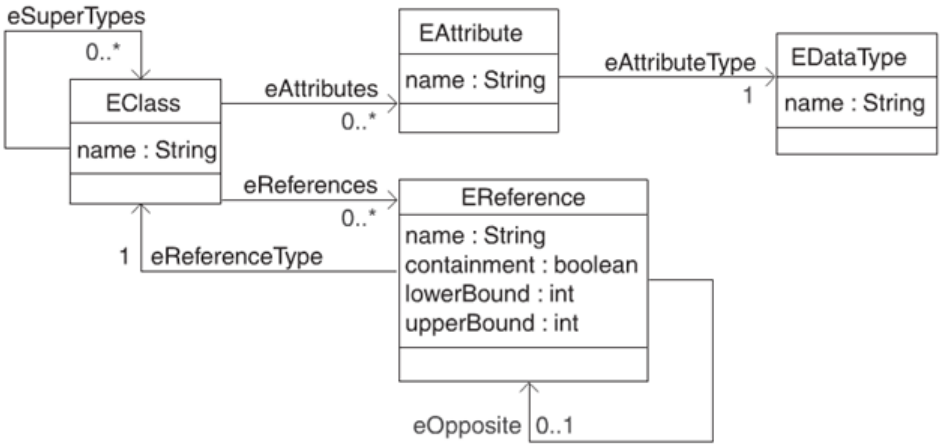

Figure 2.5: A subset of the Ecore model called *Ecore Kernel*. Taken from [SBPM09]

EClass Classes contain any number of attributes and references. A class can be a supertype of another class in order to support (multiple) inheritance.

EAttribute Attributes are the components of the classes. They are identified by a unique name within the class and they have a data type.

EDataType Data types represents the types of attributes. Primitive and object data types are supported. Ecore data types are serializable and custom data types can be created, too. The mapping between Ecore and Java is quite intuitive, for example, *EBoolean* corresponds to the primitive type *boolean* in Java or *EString* corresponds to *java.lang.String*.

EReference Associations can be modeled with references which have a type and a name, similar to attributes. Though, the reference target must be an EClass. If this EClass should also navigate to the source element, another reference must be created. For each reference, a lower and upper bound can be specified. The containment property is a strong *has-a*-relationship. The contained class cannot be part of another containment. For the opposite class, the derived *container* attribute will be true.

It should be noted that the Ecore metamodel provides a lot of other features which are not discussed here such as enumerated types (*EEnumLiteral*), behavioral features (*EOperations*), packages, factories or annotations. A complete reference can be found in [SBPM09].

2.3.2 Model Driven Architecture

Model Driven Architecture (MDA) is one approach for realizing model driven software development which was created by the OMG in 2001. It was built around a set of standards including UML, MOF, XMI and others. MDA was introduced to tackle concerns which appear during the realization of software, especially that a software system eventually has to be deployed on at least one platform. However, platforms evolve over time or must be replaced by others. That is why MDA strictly separates the high-level model from platform-specific technologies. Hence, the primary goals of MDA are portability, interoperability and reusability, achieved through separation of concerns [OMG03]. More precisely, MDA defines three viewpoints as illustrated in Figure 2.6.

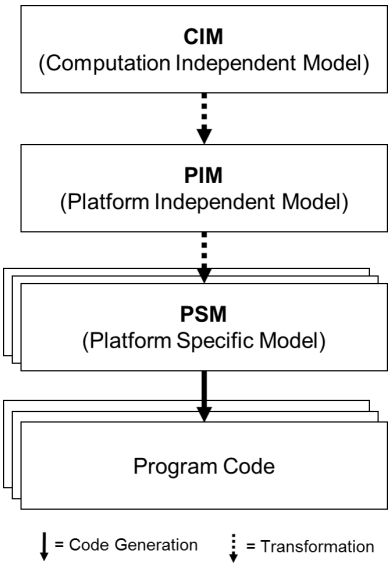

Figure 2.6: The MDA approach. According to [OMG03]

Computation Independent Model (CIM) This viewpoint contains the business environment and its requirements while hiding any details of the system's structure. A CIM describes what the system is expected to do. It is assumed that the primary creators of this model are domain experts with an excellent understanding of the functional requirements. Though, they are not familiar with realizing functionality of the domain requirements. Actually, the CIM plays an important role in bridging the gap between software engineers constructing the application and domain experts.

Platform Independent Model (PIM) Models on this layer describe the system while keeping the generality to be suitable for a number of different platforms. A common technique is to define a set of services which are defined independently of specific platforms.

Platform Specific Model (PSM) As the name suggests, a PSM models the necessary technology details which are specific for a particular platform. It unifies the specification in the PIM and details about the use of a particular type of platform.

The key challenge in MDA is to transform higher level models into models of a lower abstraction level. Eventually, executable implementation code can be generated. The process of model transformations will be discussed later (see Chapter 2.3.3).

Typically, the CIM is the first model to be developed when starting to construct a new application. Theoretically, a PIM should be generated out of a CIM. Though, Krioule et al. [KAG15] showed that their evaluated transformation methods support only a semi-automatic transformation. Their conclusion is that existing methods are not mature enough to fully support automated transformations including traceability features between CIM and PIM.

The next step is transforming a PIM into a PSM by adding technical information of the target platforms. The main function of a PSM is enabling the transformation to produce program code. It should be noted that this last transformation step is not considered as a separated viewpoint in MDA [GMB09]. Nevertheless, it is shown for clarity. Furthermore, the cascading model transformations from top to bottom can also be reverted from program code to CIM. However, this is subject to reverse engineering which is not covered in this thesis.

2.3.3 Model Transformations

Model transformation is one important technique in MDSD. It can be used for a variety of tasks such as changing, creating, adapting, merging or refining models. Völter and Stahl distinguish between two transformation types: model-to-model (M2M) and model-to-code (M2C) transformations. Typically, M2M converts a source model into a target model[9]. The transformation determines the mapping rules from the constructs of the source model to those of the target model. Both models do not have to adhere to the same metamodel but they are allowed to do so. In M2C transformations, source code of a programming language is generated out of the model transformation. In order to not restrict the code generation, all kinds of text fragments are allowed to be generated which is why this transformation is also called model-to-text transformation. M2C can be considered as a special type of M2M except that no metamodel of the target model is needed. M2C transformations are much harder to

[9] Actually, a M2M is able to handle multiple source and target models as described later. For simplicity, singular form is used throughout this thesis

analyze because of the lack of text structure [SVC06]. However, as the major tasks of this thesis are related to M2M transformations, the focus lies on this type.

Figure 2.7 illustrates the basic idea of a M2M transformation. The primary goal is to transform model A conforming to metamodel MM_a into model B which conforms to metamodel MM_b. The transformation itself can also be defined as a model being an instance of a metamodel of transformations of this type. In fact, MM_t is a representation of model transformation languages (see Chapter 2.3.4).

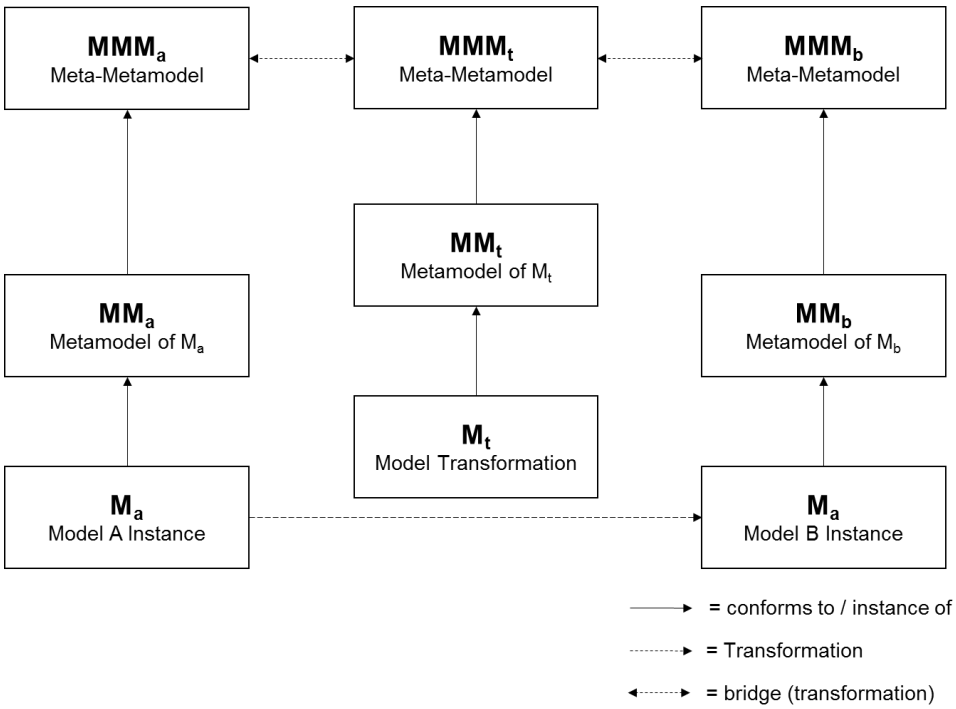

Figure 2.7: The concept of model transformations. Adopted from [JABK08]

In theory, all metamodels can potentially adhere to different meta-metamodels as illustrated. Djuric et al. [DGD06] showed that transformations between different *modeling spaces* are possible. Modeling spaces are defined as a "*modeling architecture defined by a particular meta-metamodel*"[DGD06] The following classification is crucial:

Conceptual Modeling Space Defines the abstract semantics but lack a representation (serialization) of their abstractions, for example MOF or Ecore.

Concrete Modeling Space Provide a notation but lack meaning of its syntax, for example Extended Backus–Naur Form (EBNF).

Parallel Modeling Space Categorizes modeling spaces that describe the same real-world entities in different ways. MOF and Ecore are parallel modeling spaces.

Orthogonal Modeling Space Classifies two modeling spaces of which one represents the concepts of the other modeling spaces in a concrete syntax such as XML. For instance, EBNF is orthogonal to MOF or Ecore.

If a special transformation, which is called bridge, between the conceptual modeling spaces can be defined, then their model instances could be transformed, too. The conceptual modeling space transformation itself must be developed in a conceptual modeling space which is able to represent both source and target conceptual modeling spaces. In addition, the transformation must be in a concrete modeling space which is orthogonal to source and target modeling spaces. For example, a bridge can be implemented between MOF and Ecore which both are represented by model instances of EBNF. The bridge transformation maps the concepts of one conceptual modeling space to concepts of the other one. Therefore, the bridge must be able to represent both meta-metamodels. Using UML as an example, a UML class defined by MOF will be transformed into a corresponding EClass in Ecore and so on [DGD06]. However, transformation languages used in practice such as ATLAS Transformation Language (ATL) or Epsilon Transformation Language (ETL) require that all metamodels of the transformation process must conform to one and the same meta-metamodel.

The classification of model transformations into M2M and M2C is coarse-grained. A finer division was made by Biehl [Bie10] who identified the following criteria for characterizing model transformations:

Target Type This classification introduces the distinction between M2M and M2C transformations which has already been discussed.

Change of Abstraction Source and target model could be associated on two different levels of abstraction, called *vertical* and *horizontal* transformations. Vertical transformations change the level of abstraction. They could add new details (refinement) or remove existing properties (abstraction) which are irrelevant for the target model. For example, a refinement transformation could get an abstract platform-independent model as input in order to produce a platform specific model. In contrast, a horizontal transformation only changes the view or representation of the model without modifying the level of abstraction, for example pretty-printing or refactoring. A special kind of horizontal transformations are *translation* transformations where the source model is converted to a target model of a different metamodel while the information level does not change.

Change of Metamodels Transformations of source to target model with a common metamodel are called *endogenous transformations*. On the other hand, *exogenous transformations* map the constructs of two different metamodels in order to produce a model instance of a different metamodel.

Technical Spaces Technical spaces are the working context where applications can be developed and specified from a certain perspective. For instance, a model of the Business Process Execution Language (BPEL), which is defined in an XML Technical Space, can

be transformed into Petri nets for verification [SB05]. One can classify transformations which support crossing boundaries of technical spaces and others which do not.

Number of Models M2M transformations can be categorized by the number of input and output models

one source model, no target model These transformations are called *in-place transformations*. It is assumed that source and target model are the same. Specific parts in the source model are modified during the transformation.

one source model, one target model There is one particular source and one distinct target model. Usually, the target model is empty or contains information which can be overwritten.

one or more source models, one or more target models A target model could need multiple source models and vice versa. Often, the various input models are interconnected or represent different perspectives of a common object.

Preservation of Properties Source and meta model have common properties which are not modified during the execution of the transformation.

2.3.4 Model-to-Model Transformation Languages

Model transformations can further be distinguished based on the model transformation language. The following few examples show some possibilities of classification: The language paradigm might be *imperative*, specifying a specific control flow, or *declarative*, describing a mapping relationship between source and target metamodels or a combination of both constructs, called *hybrid* approach. Transformation rules can be defined *unidirectionally* or *multidirectionally*. Unidirectional transformations can only be applied from source to target models, while multidirectional languages allow transformations from multiple source models into various target models as well as vice versa. In case, there is only one source and one target model, *bidirectional* transformations can be performed, for instance by the ATL tool [JABK08]. In fact,there are lots of other categories which are described in [Bie10].

Actually, a large number of transformation languages can be found on the market. One common standard including tool support is the Query/View/Transformation (QVT) specification [OMG15] established by the OMG. Indeed, QVT consists of three different model transformation languages which all conform to the MOF standard. *QVT Operational Mappings* is an imperative transformation language, whereas *QVT Relations* is a high level- and *QVT Core* a low level declarative transformation language. The three languages support different community requirements such as uni- or multidirectional transformations, *check-only* transformations for verification or incremental updates on models.

Another well-known transformation language, inspired by the QVT requirements, is the already mentioned ATL which is a hybrid M2M transformation language. While the declarative constructs are designed for simple model transformations, the imperative part can handle more complex ones. ATL supports only unidirectional transformations operating on read-only source models and creating write-only target models. Transformations are organized into *modules* where multiple *transformation rules* can be defined.

However, ETL will be discussed more closely because it has strong support for EMF projects as an integrated transformation language. All mapping tasks described in Chapter 5 can be accomplished with this language which is why it has been decided to not evaluate other transformation languages for the purpose of this thesis.

Eclipse Transformation Language

ETL is a hybrid M2M transformation language. In fact, it is one of multiple languages provided by the Epsilon project which is a platform for model management tasks such as model comparison, code generation, merging, validation or model transformation. Each language is supported by Eclipse-based development tools including interpreters in order to execute programs written in these languages [KRGDP15]. Actually, ETL is built on top of another Epsilon language called Epsilon Object Language (EOL).

ETL is able to transform more than one source model into multiple target models. Developers are able to query, navigate and modify source and target models during execution. However, ETL only supports unidirectional transformations. Similar to ATL, ETL transformations are organized into *modules*, containing zero or more *transformation rules*. The general syntax of a transformation rule is illustrated in Figure 2.8.

```
1  (@abstract)?
2  (@lazy)?
3  (@primary)?
4  rule <name>
5    transform <sourceParameterName>:<sourceParameterType>
6    to <rightParameterName>:<rightParameterType>
7       (, <rightParameterName>:<rightParameterType>)*
8    (extends <ruleName>(, <ruleName>)*)? {
9
10   (guard (:expression)|({statementBlock}))?
11
12   statement+
13 }
```

Figure 2.8: Syntax of a transformation rule. Taken from [KRGDP15]

A rule can be annotated as *abstract*, *primary* or *lazy*, a concept that first was introduced by ATL. Abstract rules must be extended as they only are invoked when another rule extending them is executed. In the scope of equivalent rules, primary rules are executed first. In contrast, lazy rules have to be invoked explicitly as they are not executed automatically. Each rule has a unique name within the module and specifies one source and one or more target parameters. *Guarded* rules are only executed if its expression evaluates to true. Eventually, body statements which consists of EOL expressions are executed.

In addition, developers can define *pre* and *post* blocks, invoked in the beginning of the transformation and after its execution. Except for the mentioned annotation effects, rules are executed from top to bottom. Finally, ETL transformations can be executed involving the user who could be asked for additional information during the transformation process.

3 The model-driven Approach for REST Services by Haupt et al.

As discussed before, obeying the REST constraints for complex software systems is a sophisticated task. Many services which are claimed to be RESTful by their developers violate against one or more constraints. In order to support developers to build REST compliant services or even force REST compliance by design, Haupt et al. developed multi-layered metamodels for this purpose.

As these models, especially the *Resource Model* and the *URL Model*, are central to the work of this thesis, their basic concepts will be discussed in the following. A complete reference of the approach can be found in [HKLS14] and [HLP15].

3.1 The multi-layered Metamodels

An overview of the multi-layered metamodels is illustrated in Figure 3.1. The first starting point is the *Domain Model*. It is independent of the modeling paradigm. Depending on the skills of the domain experts designing the application, the best fitting modeling paradigm can be chosen, for example, UML or entity-relationship diagrams. Domain experts express their knowledge by designing APIs on this level without the need of knowing REST. It should be noted that this model is optional since the Resource Model can be created directly.

After that, the Domain Model is transformed into an *atomic*- or *composite* Resource Model. Depending on the Domain Model, additional information could be needed in order to conduct this transformation [HKLS14]. The atomic Resource Model describes basic elements such as resources, methods or representations. In contrast, composite Resources Models are additional abstraction layers allowing to aggregate specific parts of atomic Resource Models with the aim of reducing complexity. Both models can be mapped automatically. Resource Models must be designed by a REST expert.

Service descriptions (see Chapter 2.2.2) can be derived from Resource Models. However, for the purpose of creating service descriptions of existing description languages out of Resource Models, the *URL Model* is required additionally. The reason is that the Resource Model is strictly separated from the URL structure of the REST service.

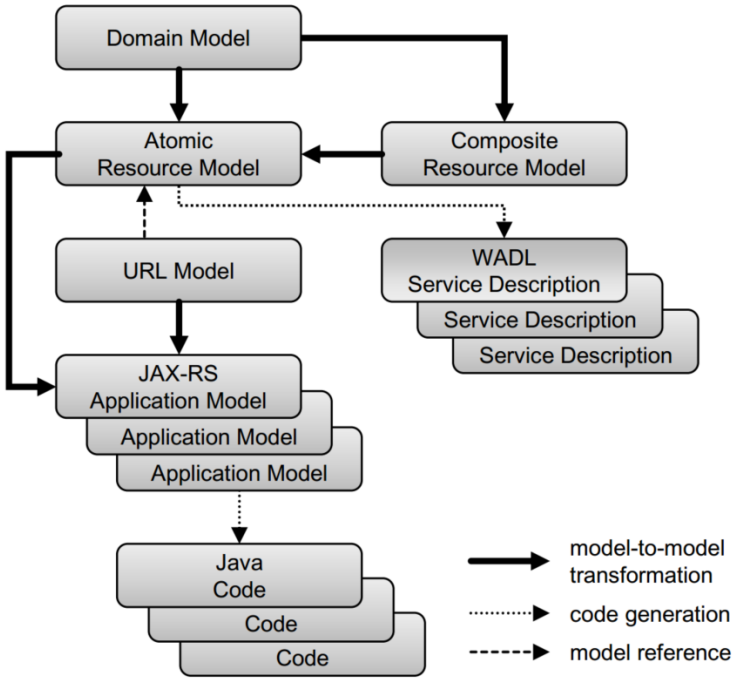

Figure 3.1: The layered metamodel for designing REST APIs. Taken from [HLP15]

URL and Resource Model are interleaved in most other REST API description languages, although in the proper sense of HATEOAS service consumers should not rely on a predefined URL structure but rather use links or other meta-data offered by the resource representation in order to interact with resources. This leads to a more loosely coupled client which does not break after changes to the URL structure. Consequently, URL information which still is required for deployment purposes, should be placed outside the actual resource description.

The URL Model takes the Resource Model as input and assigns an appropriate URL structure to it. Because of HATEOAS the clients only need to know the root resource. Though, the application provider has to build a maintainable URL structure. The URL Model is needed as input for the application models.

Until now, all discussed models are PIMs. But in order to create program code, application models (PSM) must be generated, depending on a particular platform. In general, the meta-models are independent of the target platforms. Figure 3.1 shows JAX-RS as an example target specification. Typically, the generated program code needs further refinement regarding the application logic.

Since Resource and URL Model are part of the transformation task (See Chapter 5), the following two sections analyze their realization in Ecore.

3.1.1 The initial Resource Model

Figure 3.2 illustrates the Ecore metamodel of the Resource Model. The top level class is the *Resource Diagram* which can contain multiple *resources* and multiple *links*. Resources are the central class where the four most important HTTP methods GET, PUT, POST and DELETE can be defined. Each kind of HTTP method is only allowed once per resource. Except for POST, all methods specify their media types directly as attributes of their classes. Moreover, these three methods may carry multiple *parameters* which are semantically the query parameters occurring in a URL. The POST method will be discussed later.

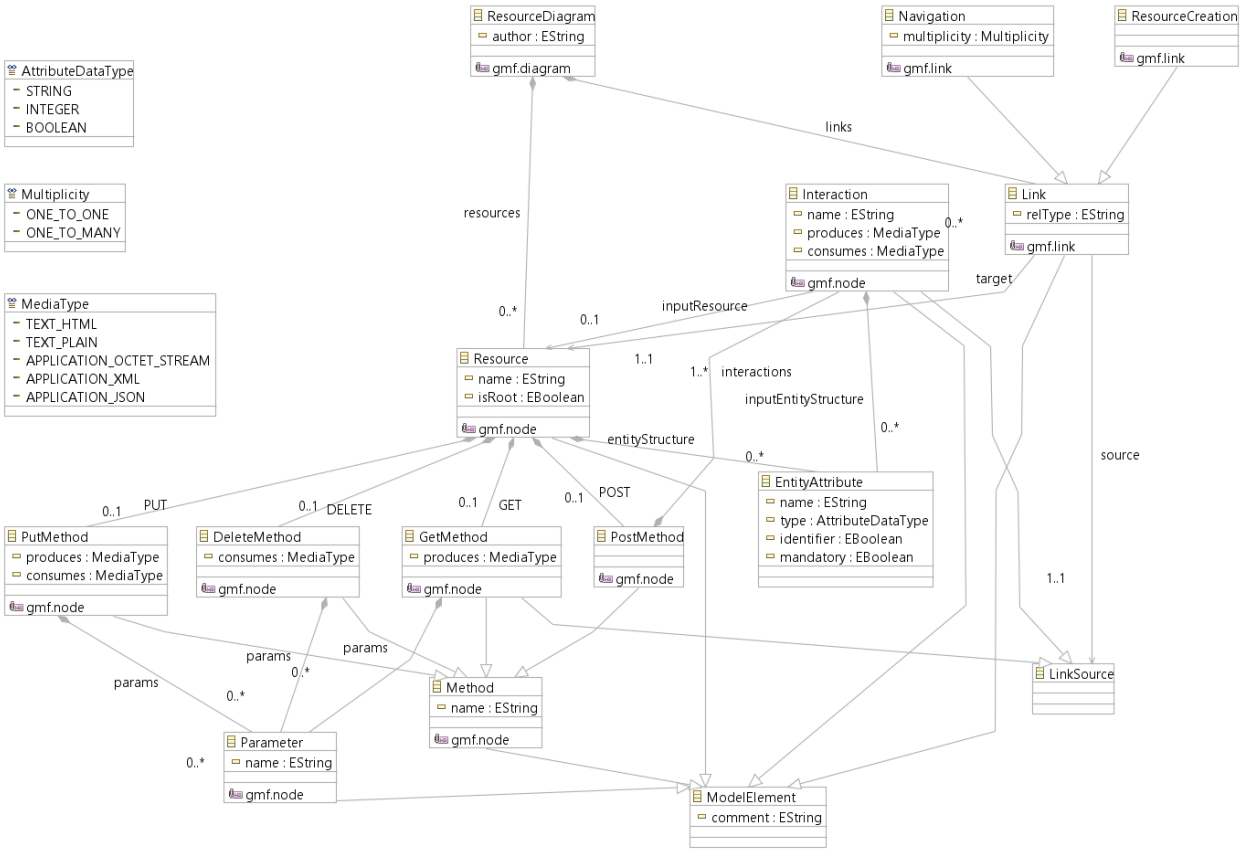

Figure 3.2: The initial Resource Metamodel

Furthermore, multiple *entity attributes* can be attached to a resource. Their meaning is to define the abstract resource with attributes which are independent of the actual representation. For example, a resource might define a person who has multiple entity attributes such as name, gender, age, etc. A GET request might retrieve this person as XML, JSON or plain text content. In order to keep the model simple, only a limited number of media types and attributes types are supported.

In contrast to the other HTTP methods, the POST method contains one or more *interactions* where name and media types can be specified. The reason for the existence of *interaction* elements is that the body schema of a POST request might not represent the structure of a resource. A POST request might carry just one part of the entity attributes or even define a completely new structure. Therefore, interactions can also have the possibility to reference to entity attributes. This is not true for GET responses and PUT requests which both should contain a representation of the whole resource.

In addition, resources can be linked with each other using the abstract *Link* class. Implementations of this class are *navigations* and *resource creation* links. As the name suggests, navigation links should be chosen when navigating from one resource to another, for example via a GET request. On the other hand, resource creation links should be used when creating a new resource via POST or PUT. In this version, only GET and POST (via interactions) can be link sources, whereas the target is always a resource.

Changes to the initial Resource Model

Some important changes were made to the initial version in order to have a profound basis for the model transformations covered in Chapter 5. For example, the media types of the HTTP methods were adapted according to the HTTP specification. The RFC 7231 [FR14c] states that "*A payload within a GET [or DELETE] request message has no defined semantics; sending a payload body on a GET [or DELETE] request might cause some existing implementations to reject the request.*" Therefore, GET and DELETE exclusively allow only *producing* media types.

In addition, some other adaptions were made to be in inline with the HTTP specification, for example, allowing PUT and DELETE to also be a *LinkSource* type since their response body could include links to other resources, too. Furthermore, all HTTP methods may define query parameters which were also extended by new fields called "mandatory" and "type".

After applying these changes on the Resource Model, the resulting version is used for the model transformations.

3.1.2 URL Model

The URL Model[1] contains one top level class called *Deployment Model* which can have multiple *Mappings*. As shown in Figure 3.3, the Mapping class contains multiple references to classes in the Resource Model: It contains an attribute called *derivedFrom* referencing to a Link class in the Resource Model and it also maintains references to source and target resource. Finally, the Mapping class defines a reference to other *URL Fragments* via the "url"-attribute. Regardless of

[1] in the model implementation, the URL Model is called "Deployment Model"

the link type, the URL Model assigns one *URL Fragment* to each link in the resource model. The URL Fragment can be either a *static* or *dynamic* URL Fragment. While the static URL Fragment is a fixed relative path, for example */user*, the dynamic URL Fragment is in fact a path parameter, for example */{id}*. The dynamic URL Fragment is specified by an entity attribute of the Resource Model.

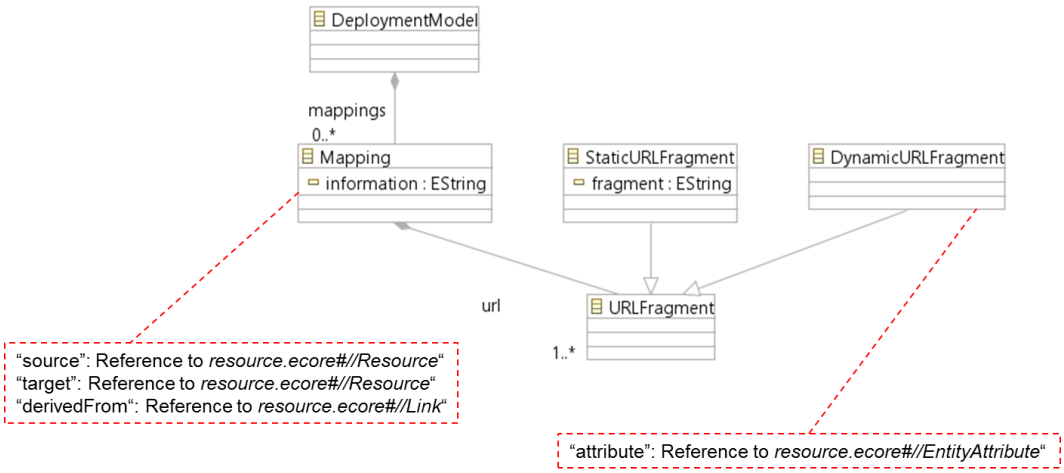

Figure 3.3: The final URL Model

3.2 Supporting REST Compliance

The described metamodels support compliance with two fundamental REST constraints: uniform interfaces and HATEOAS. The following subsections describe how they are ensured.

Uniform Interfaces

The correct use of the HTTP methods and their semantics is ensured by two aspects. First, domain experts are not responsible for defining an appropriate Resource Model. The transformation from Domain to Resource Model ensures the right use of the interfaces. Furthermore, REST experts refine and improve these models. Second, the composite Resource Model helps simplifying the Resource Model on a higher level of abstraction. As concrete modeling tasks are generated from composite Resource Models, human-caused errors are avoided.

HATEOAS

The Resource Model requires to design the relationships between resources which is the key concept of HATEOAS. Moreover, URL information is separated from Resource Model, decoupling the client from a predefined URL structure. The purpose is to realize HATEOAS which demands that interactions are driven by resource representations. Therefore, links are used to connect resources.

4 API Description Languages in Industry

A lot of propositions were made in order to describe REST APIs properly. This chapter focuses on those languages which are used in industry. First, two popular REST API DLs are identified. Then, each of the selected languages is analyzed in more detail, especially their specification. Finally, a new REST API DL called RADL will be presented.

4.1 Finding popular REST API Description Languages on the Market

In order to make a rough selection of DLs which should be taken into consideration, the first requirement is that a description language should be in active use for describing real systems. Ideally, the language is used within multiple companies. However, it is hard to find quantitative information about the use of a particular DLs in industry.

When searching the Web for “REST API Description Language” the following names (sorted alphabetically) appear repeatedly:

- API Blueprint
- I/O Docs
- Swagger
- RAML
- WADL
- WSDL 2.0

It should be noted that any research efforts defining a new REST API DL are excluded. For the purpose of getting an impression of the languages’ popularity, the following quantitative criteria are applied on these DLs:

Google Search Google results after searching for: "[name of DL]" + "REST".

Stackoverflow Questions (in Title) Number of questions on this platform with a question title containing the name of the DL.

Total Github Projects Number of all Github projects which contain the name of the DL in their project name.

Github Spec Starred/Forks Excluding WADL and WSDL 2.0, all language specifications are published on Github. This metric measures how many people have "starred"[1] and forked this project.

Table 4.1 illustrates some general information about each language before listing the results of applying these evaluation criteria on each DL. The analysis was conducted on 04/11/2015. I/O Docs has the lowest rating in all categories. As this DL seems not to be relevant on the market, it will not be considered further. In a similar way, WSDL 2.0 has the second lowest rating in the categories *Google Search* and *Stackoverflow Questions*. In combination with REST services, WSDL 2.0 has not been adopted widely.

Table 4.1: API Description Languages Overview

	API Blueprint	**I/O Docs**	**Swagger**	**RAML**	**WADL**	**WSDL 2.0**
Company/ Sponsor	Apiary	Mashery	Reverb	MuleSoft	W3C	W3C
First Release	April 2013	July 2011	July 2009	Sep. 2013	Aug. 2009	June 2007
Format	Markdown	JSON	JSON	YAML	XML	XML
Google Search ("[name]" + "REST")	27k	4k	860k	86k	88k	14k
Stackoverflow Questions (in Title)	49	2	1,026	67	154	23
Total Github Projects	269	34	1,741	501	168	-
Github Spec Starred/Forks	2865/ 844	1,646/ 408	2,259/ 720	1,962/ 123	-	-

[1] adding a project to someone's favorites

As a matter of fact, Swagger dominates in all categories. The relatively small amount of Github stars/forks is related to the fact that Swagger has many other repositories which are even more popular than the Swagger Spec, for example the Swagger UI repository with over 3000 *stargazers* as they are called in Github.

After eliminating two obviously inferior options, the third elimination is more difficult. WADL, RAML and API Blueprint share similar results. Among these three DLs, WADL has most Google Search hits and Stackoverflow questions whereas RAML has the most Github projects, while the specification of API Blueprint was starred and forked most often.

As a distinct decision cannot be made based on this table, other comparisons of REST API DLs are taken into account. Heritage [Her14] states that the most used API DLs for REST are Swagger, RAML and API Blueprint. She points out that the first two are most widely adopted in enterprises. Actually, this is the predominant opinion of many other Blog authors, for example [Sto14] or [SAN15]. Thus, for this thesis the ranking Swagger before RAML before any other REST API description language is set.

The category *community size* of Swagger gets the highest rating in the comparison by Stow [Sto14] which is in line with Table 4.1. Furthermore, the authors make a qualitative analysis of multiple aspects such as build support, usability, tooling, etc. Stow rates the specification of WADL as complex and time consuming with only little tool support for designing top-down descriptions. Most often, WADL is just generated out of JAX-RS implementations such as Jersey.

Recently, a consortium of powerful enterprises such as Google, IBM, Paypal, etc. founded the Open API Initiative(OAI)[2]. Its purpose is to standardize the description of REST APIs. The goal of this initiative is to create, evolve and promote a vendor neutral API description format which is based on the Swagger specification. According to their website, the reason for taking the Swagger specification is because "*with 350,000 downloads per month [...], Swagger is the word's most important description format for defining RESTful APIs*".

Consequently, for the two leading REST API DLs Swagger and RAML, transformations to the Metamodel by Haupt et al. and vice versa will be implemented. Originally, the third choice would have been API Blueprint which a conceptual mapping to the Resource/URL Model should be established for. Though, after some research on API Blueprint's specification it turns out that it does not differ much from Swagger's metamodel. By the way, the same is true for I/O Docs. As they all share similar metamodels it is assumed that a separated conceptual mapping would not bring much more valuable insights. This is why a different selection is made: The relatively new REST API DL called RADL is chosen as the third language because it follows a completely different approach compared to the others (see Chapter 4.2).

[2]https://openapis.org/

4.2 Analysis of selected API Description Languages

The following chapter is devoted to give a more detailed background on three selected REST API DLs identified previously. Besides the specification of machine readable definitions, a crucial aspect for adoption is its tool support throughout the API design lifecycle. Of course, each DL has its own tooling. However, instead of presenting vendor-specific tools, some general building blocks have been identified which are supported by most providers in some way. The following list summarizes the most important tools.

Parser & Code Generators For the purpose of programmatic generation and consumption of description files, parsers and code generators are essential to every DL. Usually, they are available in multiple programming languages. Code generators can be further distinguished into server side and client side code generators. Server side code generators create a code skeleton out of the API description, whereas client side code generators build client libraries for consuming the API.

Editor/Workbench An IDE environment is crucial in order to support the developer with designing new descriptions or editing existing ones. It should provide multiple features to ease the development. For example, syntax highlighting or visualizing the code in real time into a human-friendly documentation view as it is done by the Swagger Editor.

Validation Clearly, there must be an opportunity given to validate an API definition. Ideally, a validation service is already integrated into the editor. Additionally, another type of validation is automatically checking API calls against the interface definition.

Interactive Documentation A human-friendly documentation should be generated out of the definition. In addition, the documentation could also provide testing capabilities as it is the case in the *Swagger UI*.

Translator For most description languages there is a tool for translating between different API definitions, e.g. Swagger2RAML although their metamodel is different. Actually, the idea behind these tools is the main work of this thesis presented in Chapter 5.

4.2.1 Swagger

Swagger, which has been started as a closed source project in 2009 by the company *Reverb* (formerly *Wordnik*), consists of an extensive specification and a framework implementation for describing, producing and visualizing RESTful Web services [Swa15]. Meanwhile Swagger has evolved into a public, widely-used project, driven by its community.

The specification, which is now the base of the Open API Initiative, and all public tools are licensed under the Apache 2.0 License. The rich tool support of Swagger allows a contract-last as well as a contract-first approach since Swagger does not prioritize one technique for its API

development. The Swagger definition file can be written in JSON or YAML. Both formats are interchangeable as they support the same specification. At the time of writing, the current release version is Swagger 2.0, officially released in September 2014. This version is used for the work of this thesis.

The Swagger 2.0 specification [Swa14] is a highly detailed, informal description of Swagger's metamodel. In fact, the specification also includes a corresponding formal metamodel described in JSON Schema. Though, in order to get an overview of the general structure of the metamodel, Figure 4.1 visualizes the basic elements in an Ecore model. It is important to note that some language features have been omitted for clarity reasons.

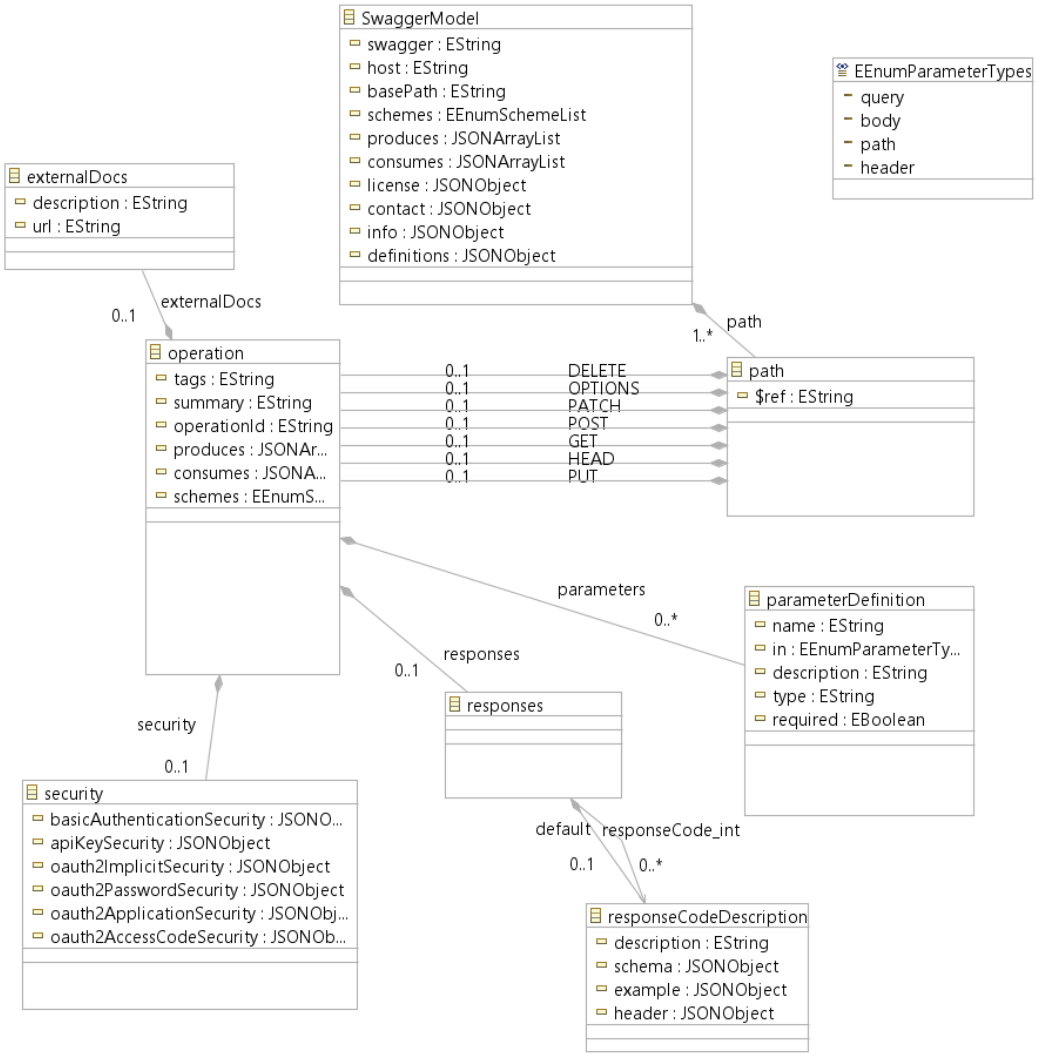

Figure 4.1: Simplified Swagger Metamodel

The Swagger model defines some general information on the root level, for example the *Swagger version*, the *host* containing the host name or IP where the API is served at and its *basePath* which is relative to the host. In addition, a mandatory *info* object contains other general information such as title, description, API version, contact, etc. Moreover, the global *schemes* property defines the transport protocol such as HTTP(S). The fields *consumes* and *produces* define a global list of media types in order to define incoming and outgoing messages in respect to their MIME types. Some global properties can be overridden within the *paths* definition which will be explained later. Although it is not a required field, nearly all Swagger descriptions contain a *definitions* section. It is a way of describing reusable schemas being referenced from multiple locations. It is useful for defining the structure and the data types of parameters, responses, security definitions, etc. The JSON schema reference (the property is called *$ref*) is used for referencing to the according *definitions* sections.

The most essential root object is the *paths* object containing multiple *path* objects to single resources. Each path contains at least one operation which is identified by its HTTP method. Obviously, only at most one operation can be defined for each HTTP method within one path. Each operation, regardless of the HTTP method, is treated the same way. It contains some general properties such as a summary or the possibility of overwriting the previously defined global properties schemes, produces and consumes. Furthermore, each operation may define multiple parameters. Besides the standard fields such as parameter name, (data) type, description and marking the parameter as required, Swagger has a special notation for the different types of parameters which will be explained in the following.

query These parameters are appended to the URL, for example */persons?id=123*. The query parameter is *id* with the value 123.

body The purpose of the body parameter is to define the schema of the payload of an HTTP request. That is why there can only be one body parameter.

path This type of parameter specifies the parameter within the operation's URL using path templating. For example the URL template */persons/{personId}* includes the path parameter *personId*. In case of the URL */persons/5*, the value of the path parameter is 5. It should be noted that the data type is specified in the parameter definition.

header Expected custom header fields as part of the request header can be specified with this type of parameter.

formData This type of parameter requires that the content type of the request (the consumes-property) must be either *application/x-www-form-urlencoded* or *multipart/form-data*. Form parameters are sent in the payload which is why *formData* parameters cannot be declared together with body parameters.

It should be noted that there is in fact just one parameter section where all multiple parameters can be defined. Figure 4.1 defines the zero-to-many cardinality of the parameters within the parameter section.

Furthermore, each operation is required to define a response object which must contain at least one response definition. This could be either a default response or a specific response code which is identified by an Integer, for example 200. Most response objects contain a schema object referring to a specific definitions section. However, it is also possible to define the schema inline or as an array schema. The response schema determines the schema of the payload which will be sent to the client as a response to the corresponding request.

If external documentation is needed, it can be defined as part of the operation but also within other objects. Finally, Swagger supports security features such as basic authentication, API-key or OAuth(2). Security definitions can be globally defined or within the HTTP operations. Each security definition has a different schema in order to get the required properties, for example OAuth security definitions need a *token URL* which is not needed for basic authentication. Listing A.4 shows a minimal Swagger example defining all required fields according to the specification.

4.2.2 RAML

RAML is an acronym standing for **R**ESTful **A**PI **M**odeling **L**anguage. A dedicated RAML Workgroup, consisting of multiple companies such as MuleSoft (leader), VMware, Akana Software, Cisco and others, aims to build an open specification, including rich tool support, for describing APIs to further enhance the potential of the API economy [RAM15b]. RAML and its tools are open source. The RAML specification 1.0 states that RAML "*provides mechanisms for the definition of practically-RESTful API's*" [RAM15a]. The support for HATEOAS is limited.

In contrast to Swagger, RAML is a pure design-first approach. RAML's focus lies on designing clean and readable descriptions which is why they have chosen YAML as their design markup language. Due to the human-centered approach, RAML has a strong support for reusable patterns, namely *resource types* and *traits* which will be discussed later. The RAML community claims that the biggest strength of RAML is that those descriptions can be understood by non-technical stakeholders, too. Its about the *vision* behind the API definition and not about the implementation. On the other hand, Swagger was developed in order to solve a workflow problem: First, its goal was to document existing APIs accurately and provide client tooling for Swagger descriptions. Swagger's design-first approach came afterwards [Lan14].

The latest RAML specification is version 1.0. However, at the time of implementing the model transformations (see Chapter 5.2) the latest version 0.8 was used. Compared to the older version, specification 1.0 has only a few changes. However, it is not fully backwards compatible: Some new features (e.g. annotations, libraries, etc.) are added while other existing constructs have been changed, for example the data model, the concept of named parameters or some naming conventions (*baseURL* instead of *baseURI*). The following paragraph focuses on specification 0.8 while mentioning features that have been changed in version 1.0.

In fact, the fundamentals of RAML's metamodel do not differ much from Swagger's metamodel. The following list categorizes some main features of both languages while emphasizing their differences if present.

Basic Information (root) Version, title, media types, etc. can be defined here. Unlike Swagger, external files such as schema files, resource types or traits (see later) can be referenced. This means that there are usually multiple files defining an API.

Schema Definition Swagger supports only a subset of JSON Schema. Even if defined in YAML, there are some restrictions. This is a major difference since RAML is much more flexible. Primarily, the users should use the YAML based data modeling language but they could also include native XML or JSON schemas.

Resources RAML resources are essentially the same as Swagger paths. The major difference is that RAML resources can be nested and that RAML can define resource types.

Methods There is no difference in the definition of RAML methods and Swagger operations, except that security schemes are referenced in RAML with "secured by" and the application of traits.

Parameter Types Parameter types are the same as in Swagger, except there is no *formData* parameter in RAML 1.0, while the construct existed in version 0.8.

Response The definition of responses is essentially the same in RAML and Swagger.

Security There are some minor differences in the security definition which will not be discussed as they are not relevant for this thesis.

Resource Types and Traits

Resource types and traits are features for capturing API patterns that can be reused within and across RAML definitions. This is a unique characteristic among the top REST API DLs, the RAML community is very proud of [Amu15b]. According to Uri Sarid, one founder of RAML, resource types and traits improve readability and consistency among multiple APIs, it reduces complexity for both servers and clients and it speeds up the development process. Best practice patterns could emerge that multiple providers could use for their different APIs.

Basically, a resource type is a partial definition of a resource. It can define methods, parameters, security schemes as any other resource, too. An actual resource that uses a resource type inherits all its properties. In fact, resource types may also inherit from another resource type, leading to an inheritance chain. RAML resources may use zero or more resource types by defining their *type*-property. In general, resource types can be compared to abstract base classes.

Traits are similar to resource types except of their scope. While resource types are applied on resources, traits are partial method definitions. They provide method-level properties such as descriptions, query string parameters, responses or headers. Resource types could also use traits which then apply to the resource using that resource type. Traits can be applied to a resource method by specifying its *is*-property.

Listing 4.1 shows a RAML definition example where first a resource type and two traits are defined. Afterwards a resource with the path */books* inherits the resource type and uses the defined traits in its GET method before specifying a response which is defined in the same way as in Swagger descriptions. Usually, traits and resource types are defined in external RAML files.

```
#%RAML 0.8
title: Example API
version: v1
resourceTypes:
  - searchableCollection:
     get:
       queryParameters:
         <<queryParamName>>:
           description: Return <<resourcePathName>> that have their <<queryParamName>> matching
         <<fallbackParamName>>:
           description: If no values match, use <<fallbackParamName>> instead
traits:
  - secured:
     queryParameters:
       <<tokenName>>:
         description: A valid <<tokenName>> is required
    paged:
     queryParameters:
       numPages:
         description: The number of pages to return, not to exceed <<maxPages>>
/books:
  type: { searchableCollection:{queryParamName: title, fallbackParamName: digest_all_fields}}
  get:
    is: [ secured: { tokenName: access_token }, paged: { maxPages: 10 } ]
    description: |
      Get a list of books
    responses:
      200:
        body:
          application/json:
            schema: !include someSchema.json
```

Listing 4.1: Modified Example of a RAML Description. Taken from the RAML Spec [RAM15a]

4.2.3 RADL

RADL, an acronym standing for **R**ESTful **A**PI **D**escription **L**anguage, is an XML-based description language created by three employees[3] of the EMC Corporation in 2013 and was open-sourced in January 2014 [RSW15]. According to J. Robie, they started to build their own REST API DL after they realized that no existing language would satisfy their needs. Although the RADL project provides first tooling support such as generating Java code from RADL, extracting RADL descriptions from Java Code, documentation or integration with build systems, the project cannot be compared to the rich tool support of the leading REST API DL providers.

Currently, RADL is quite unknown in the market of REST API DLs which is why it is not listed in Table 4.1. One reason could be that the RADL project still is in a heavy development phase so that popularity could rise after releasing a mature version. Up to now, only a few other Web sources publicly available, e.g. the presentation on the XML Amsterdam Conference [Sin13], make RADL a subject of discussion.

RADL supports contract-first and contract-last approaches with corresponding tooling. In fact, the authors of RADL further categorize the contract-first approach into three variations: *Requirements-driven* means starting with (state machine) diagrams capturing the interactions, whereas *schema-driven* implies starting the data schema and finally beginning with examples for explaining the data structure is called *examples-driven*.

In contrast to the other description languages mentioned, RADL focuses on describing truly hypermedia-driven REST APIs. RADL is built based on states and transitions. Since hypermedia APIs can be expressed in the form of finite state machines accurately, this approach seems reasonable. The pseudo example[4] shown in Listing 4.2 should give an idea of RADL defining the most important constructs. The example RADL document describes a simplified situation in a restaurant. The client reaches the state *Menu* by firing a GET request on the resource */menu*. Then, the costumer gets a virtual menu card (through the property-group "menuProperties"), having two options in this state: He can either order something following the transition "Buy something" by making a POST request on */menu/order* or he could just follow the transition "Leave" by firing a DELETE request on the resource */menu*.

[3] Jonathan Robie, Rémon Sinnema and Erik Wilde
[4] a complete example can be found in Listing A.1

```
<service xmlns="urn:radl:service" name="ServiceName">
      <states>
            <start-state>
                  <transitions>
                        <transition name="Entry Point" to="Menu">
                  <transitions>
            </start-state>
            <state name "Menu" property-group="menuProperty">
                  <transitions>
                        <transition name="Buy something" to="Order">
                        <transition name "Leave" to "Cancel">
                  <transitions>
            </state>
            <state name="Order" <!--define transitions for next actions.. --> />
            <state name="Cancel"/>
      </states>
      <link-relations>
            <link-relation name="http://relations.someShop.com/entry">
                  <transitions>
                        <transition ref="Entry Point"/>
                  </transitions>
            </link-relation>
            <!--define link-relations for "Buy something" and "Leave" -->
      </link-relations>
      <media-types>
            <media-type name="application/ld+json">
                  <specification href="http://www.w3.org/TR/json-ld/"/>
            </media-type>
      </media-types>
      <conventions>
            <uri-parameters>...</uri-parameters> <headers>...</headers> <!--and more... -->
      </conventions>
      <resources>
            <resource name="Menu">
                  <location uri="/menu/"/>
                        <methods>
                              <method name="GET">
                                    <transitions>
                                          <transition name="Entry point"/>
                                    </transitions>
                                    <response>
                                          <representations>
                                                <representation
                                                      media-type="application/ld+json"/>
                                          </representations>
                                    </response>
                              </method>
                              <method name="DELETE">
                                    <transitions>
                                          <transition name="Leave"/>
                                    </transitions>
```

```
51                              </method>
52                        </methods>
53            </resource>
54            <resource name="Order">
55                  <location uri="/menu/order"/>
56                        <methods>
57                              <method name "POST">
58                                    <transitions>
59                                          <transition name="Buy something"/>
60                                    </transitions>
61                                    <response <!--response definition... --> />
62                              </method>
63                        <methods>
64            </resource>
65      </resources>
66      <errors>
67            <error name="http://errors.restbucks.com/missing-item" status-code="400">
68                  <documentation>No menu item found.</documentation>
69            </error>
70      </errors>
71      <property-groups>
72            <property-group name="menuProperties" uri="http://schema.org/Product">
73                  <property name="name" uri="http://schema.org/name"/>
74                  <property name="size" uri="http://schema.org/height"/>
75            </property-group>
76      </property-groups>
77 </service>
```

Listing 4.2: Pseudo Example of a RADL document

In the following, the important RADL elements are described. The root element is ***<radl:services>*** containing all other elements listed below.

<radl:states> Based on the HATEOAS principle, the client states can be expressed within this element. States may define *property-groups* (see later) and transitions to other states. As states do not indicate how they are represented in a physical document, they can be reused in multiple concrete documents. The initial state, where the client starts from at some well-known URL, is called the *start-state*. Listing 4.2 defines a starting state where it is pointed to a "*Menu*" state. In turn, this state holds an attribute referencing *property-groups* (see later).

<radl:link-relations> In order to define the semantics of a transition, *link-relations* are used. They specify how the current context is related to another resource.

<radl:property-groups> These are data models of RADL. A property-group references to a schema. Moreover, it could contain other property-groups or single properties. A schema can be defined inside the RADL file or referencing to an external source.

<radl:media-types> Media types specify the concrete representation of a state. Besides the media-type name, the developer can point to its formally defined specification. Default media-types can be specified, too.

<radl:service-conventions> The authors of RADL argue that it is a anti-pattern to specify allowable headers, URL query parameters or URL path parameters[5]for each endpoint. Rather they define service conventions globally that can define these constructs. Service conventions can be used by resources or media types.

<radl:errors> Similar to service-conventions, RADL defines error status codes in response messages globally rather than for each method of a resource.

<radl:resources> Services are implemented by resources. Actually, the resources element is very similar to Swagger or RAML paths. A resource has a name and provides the location URI. It specifies HTTP methods available for the resource. The methods reference to state transitions making them concrete for a physical resource. Finally, one method refers to media type, error code, authentication and service-convention definitions.

<radl:authentication> RADL provides constructs for defining authentication, e.g. authentication types, identity-provider, authentication-conventions or scheme-parameter can be specified with this element.

Some details have been left out. For example, the ***<radl:documentation>*** element allows developers to provide further information within each element discussed which is targeted to be read by humans. Documentation can be plain text or marked up in HTML which is useful for generating an HTML documentation for humans.

After the basic elements have been discussed, Figure 4.2 illustrates a simplified metamodel of RADL, showing the relationships between the elements. In fact, two clusters can be identified: On the left hand side, the elements responsible for supporting the HATEOAS constraint can be found: States, transitions and link-relations. The semantics of transitions is specified by link-relations, while one link-relation could define the semantics of multiple transitions. The transitions refer to states. Both, transition and states may refer to property-groups.

The model elements on the right hand side are for describing the concrete resources. These elements are well-known from the RAML or Swagger metamodels. Both parts are connected via the *transition reference*, pointing from *Methods*[6] to transitions.

[5]RADL calls them uri-template variables
[6]These are the HTTP methods where no distinction is made

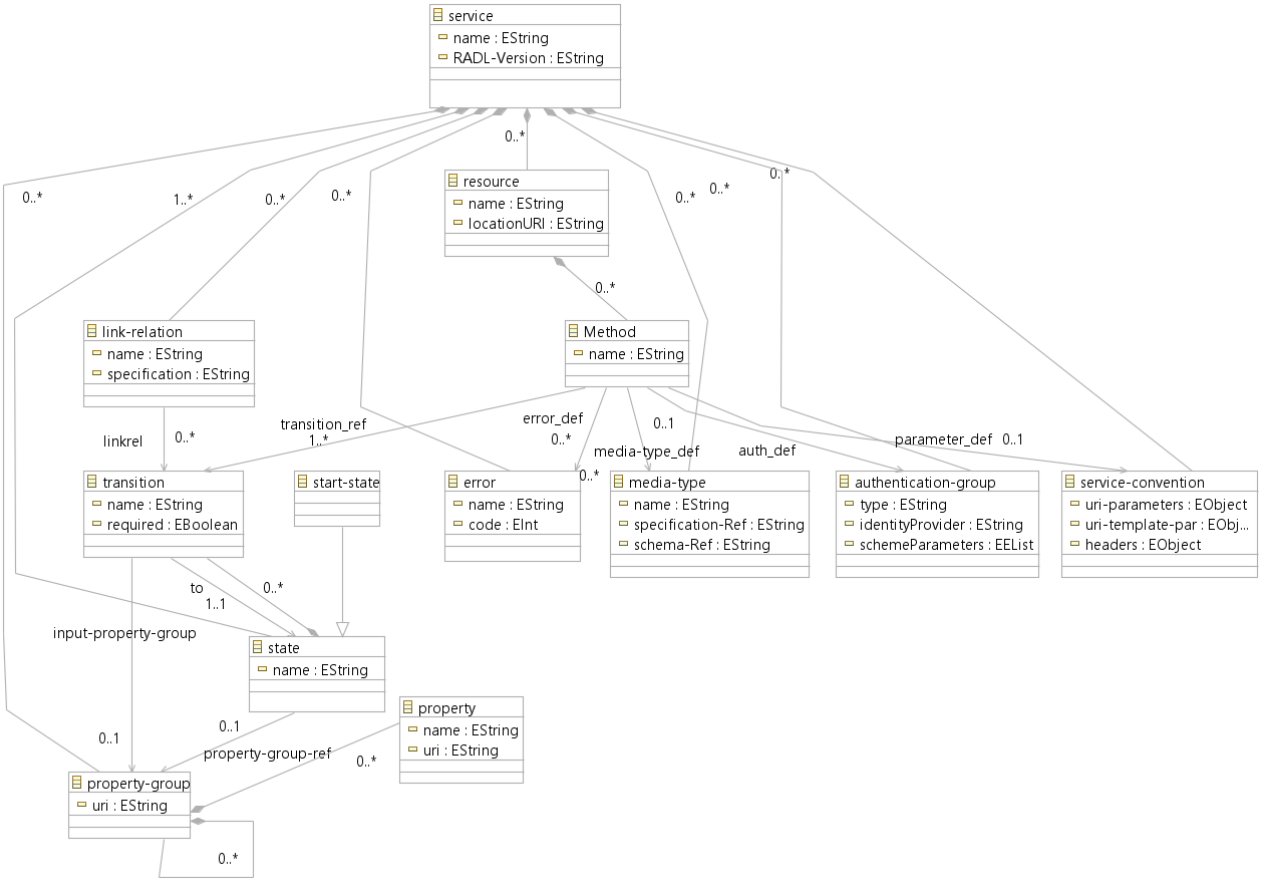

Figure 4.2: Simplified RADL Metamodel

5 Conceptual Design of Bidirectional Model Transformations

The main work of this thesis is the design and implementation of model transformations between two industry REST API DLs and the Metamodel by Haupt et al.[1].Especially the Resource and the URL Model are relevant for this work. As revealed in Chapter 4, Swagger and RAML will be the API DLs of choice. The focus lies on explaining the conceptual design of the mapping rules between Haupt's Metamodel and Swagger. Since RAML's Metamodel is equal in many respects to Swagger, only deviating aspects are discussed. Finally, the conceptual mapping between RADL and Haupt's Metamodel will give another point of view.

5.1 Model Transformation between Swagger and Haupt's Metamodel

Figure 5.1 illustrates the conceptual mapping between the Swagger and the Resource/URL metamodel. Actually, the URL model is optional[2] as Swagger does not support links between its resources. The mapping of the core concepts of these metamodels is visualized without showing any details. For a more granular view about these metamodels see Figures 4.1, 3.2 and 3.3. The next section describes the mapping rules between both metamodels before some special processing issues are explained.

5.1.1 Description of the Mapping Rules

The following paragraphs provide a description of all mapping rules which are called Swagger-Resource-Mapping (SRM) from now on. Each mapping is explained by providing a high-level description at the beginning. After revealing the reasons for the mapping approach, the implementation realized in ETL is described further. In fact, the explanation for each SRM is divided into two parts: The first part explains the transformation from Swagger to the Resource Model/URL Model (Swagger2Resource), whereas the next part outlines the backwards transformation from the Resource/URL Model to Swagger descriptions (Resource2Swagger).

[1] from now on, this metamodel is called "Haupt's Metamodel"
[2] the user decides if links should be generated

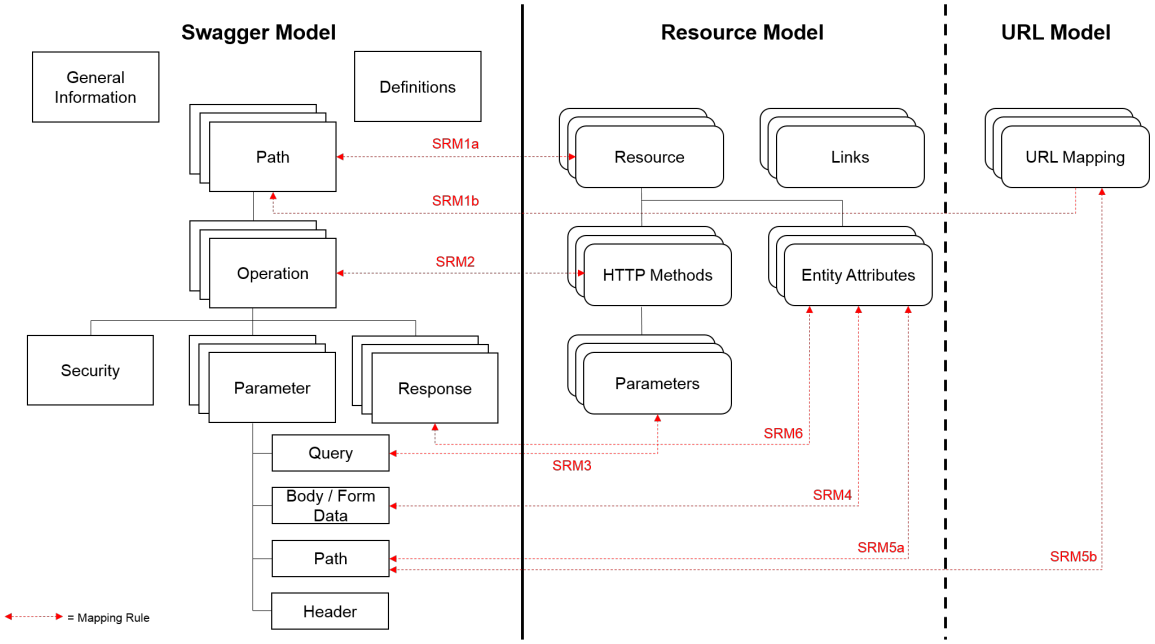

Figure 5.1: Mapping between Swagger and Resource/URL Model

SRM1: Path - Resource/URL Mapping

The first mapping describes the transformation of a path object in Swagger and a *resource* in the Resource Model. In fact, Swagger does not provide the concept of resources which are independent of any URL information. The name of the Swagger path is a globally unique identifier for the resource and simultaneously exhibits the relative path to this resource, based on the *baseURL*. Since Swagger paths and resources of the Resource Model both reference to their operations (namely their HTTP methods), the mapping is reasonable.

Listing 5.1 illustrates how Swagger paths are transformed into resources of the Resource Model (SRM1a). The ETL rule (Line 1-3) creates a root element called *ResourceDiagram* of the target model. The method *getPaths()* on the Swagger object (Line 6) returns a map of Swagger paths whose keys contain the relative URLs of the Swagger resources. For each Swagger path, a new resource of the Resource Model is instantiated carrying the name of the Swagger path. After some further processing of that new resource, it will be added to the target model, eventually.

Actually, the corresponding backwards transformation can be performed in two ways, depending on the type of the resource: Base resources do not have a matching URL mapping. In this case, the names of the resources of the Resource Model are assumed to be the URLs (SRM1a). Otherwise, the URL information is extracted from the previously generated URL Model (SRM1b). The code snippet in Listing 5.2 shows both mapping types.

```
rule Swagger2ResourceModel
      transform swagger : Source!Swagger
      to target : Target!ResourceDiagram {

      // relative URLs of all swagger paths
      var swaggerPathNames := swagger.getPaths().keySet();
      for(swaggerPathName in swaggerPathNames){
            var resource = new Target!Resource;
            resource.name = swaggerPathName;

            // [code omitted] See SRM2

            target.resources.add(resource);
      }
}
```

Listing 5.1: Swagger2Resource for SRM1

```
rule Resource2Swagger
      transform d : Deploy!DeploymentModel
      to t : Target!Swagger{

      // process base resources: Get all resources which are targets of any link
      var targetLinkResources = Source!Link.allInstances().select(link|
          link.target.isDefined()).target;
      var allResources = Source!Resource.allInstances();
      /* the variable baseResources is a set containing either all resources
      if there are no links ( and no URL model) or a subset of resources
      which are not targets of any other link --> they are base (root) resources */
      var baseResources = allResources.removeAll(targetLinkResources);

      // create Swagger paths with name of the resource
      // [code omitted]

      // process the URL information of the URL Model
      var mappings = Deploy!Mapping.allInstances();
      for(mapping in mappings){
            // mapping.source and mapping.target are the resources which are linked together
            // Swagger definitions expect absolute paths
            var absolutePath = recursivePathNameDefinition(mapping);
            // create Swagger paths with the name of the absolute path variable
            // [code omitted]
      }
}
```

Listing 5.2: Resource2Swagger for SRM1a and SRM1b

The first block of the rule (Line 5-14) processes the base resources. They are identified by not being a target resource of any link. Alternatively, if there are no links in the Resource Model, all resources are processed within the first block. Basically, a Swagger path for each resource is created with the name of the resource itself.

In contrast, the second block (Line 16-24) handles the linked resources and their URL information. The major difference to the first block is that the information of the path name is located in the URL Model. Moreover, the URL fragments only contain the path name which is relative to their superior resource, whereas the Swagger description requires path names to be relative to their *baseURL*. Therefore, a recursive method called *recursivePathNameDefinition(mapping)*, which can be found in the implementation code, was defined in order to traverse all superior resources of a given target in order to calculate the path names as expected by Swagger.

SRM2: Operation - Method

Obviously, Swagger operations can be mapped to the HTTP methods of the Resource Model as both concepts define which HTTP methods can be applied on a resource. While both metamodels prescribe that any HTTP method can be defined at most once, they differ in some points: First, Swagger allows to model all available HTTP methods, whereas the Resource Model currently defines the four most important ones. Therefore, the transformation is restricted to GET, PUT, POST and DELETE operations.

Second, the Swagger metamodel uses one operation model for all HTTP methods. In contrast, the Resource Model makes a finer distinction with regard to consuming and producing media types, conforming to the HTTP specification. For instance, DELETE and GET operations can only produce a list of media types but they cannot consume them. This is because GET and DELETE requests should be sent without body, according to the HTTP specification. Basically, Swagger allows to design descriptions which contradict to the HTTP specification. As a consequence, the transformation from Swagger to the Resource Model might loose some media type or body schema definitions which should not have been defined in the first place.

Finally, a special method within the Resource Model is the POST method as it contains an element called *interaction*, which could carry multiple, separated entity attributes . The concept of interactions was not introduced in the Swagger model. Though, input entity structures of a POST interaction are mapped to input body schemas in Swagger. However, Swagger allows only one single body schema definition. Consequently, multiple interactions on one POST method in the Resource Model will lead to loosing all except one when transforming to Swagger.

In turn, if Swagger's POST method defines a body schema which is different from that of the resource's schema, this POST method schema will be transformed into an entity structure of an interaction (see Chapter 5.1.2).

```
1       //GET method processing
2
3       var get = paths.get(key).getGet();
4       var getMethod = new Target!GetMethod;
5             getMethod.comment = get.getDescription();
6       if(get.getProduces().isDefined()){
7             getMethod.produces = processMediaTypes(get.getProduces());
8       // in Swagger, the produces property can also be defined globally.
9       }else if(swagger.getProduces().isDefined()){
10            getMethod.produces = processMediaTypes(swagger.getProduces());
11      }
12      processParameters(get.getParameters());
13      processResponse(get.getResponses());
14      resource.GET = getMethod;
```

Listing 5.3: Swagger2Resource for SRM2

Code Listing 5.3 shows a method transformation from Swagger to the Resource Model of the GET method. First the Swagger object containing the GET operation is retrieved (Line 3) and then a new GET method of the Resource Model is created (Line 4). Only the *produces* property will be transformed for this operation since GET should not consume a message body. The method *processMediaTypes()* converts a Swagger media type into a Resource Model media type, if supported. After processing its parameters and responses which will be explained in the following SRM rules, the GET method of the Swagger model is assigned to the resource of the Resource Model to be its GET method (Line 14). In a similar way, POST, PUT and DELETE are processed, too.

```
1       var swaggerPath = new Target!Path;
2       if(resource.GET.isDefined()){
3             var getOp = new Target!Operation;
4                   getOp.setProduces(resource.GET.produces.name.flatten());
5                   opn.setDescription(method.comment);
6             // parameter & response processing
7             swaggerPath.setGet(getOp);
8       }
```

Listing 5.4: Resource2Swagger for SRM2

The transformation from the Resource Model back to Swagger (Listing 5.4) is straightforward. After creating a path object of the Swagger Model (Line 1), it is checked for defined methods on a resource of the Resource Model. If a GET method is specified, media types, descriptions, parameters and responses are processed and finally attached to the Swagger path object. As mentioned before, the procedure is similar for the other three methods, too.

SRM3: Query Parameter - Parameter

In both metamodels there is the possibility to define parameters for HTTP method operations. While Swagger defines multiple parameter types, the Resource Model only specifies one parameter type, in addition to entity attributes. Actually, the semantics of this parameter type corresponds to the parameter type *query* of the Swagger model.

The transformation simply maps name, type, description (comment) and the boolean field "required" (mandatory) of both parameter types. Listing 5.5 shows this process for the Swagger2Resource transformation. Vice versa, the code is similar and trivial which is why it is omitted. Though, there is the general problem of nested objects which cannot be mapped accurately (see Chapter 5.1.2).

```
1       for(parameter in path.getParameters()){
2             if(parameter.getIn()=="query"){
3                   var modelParam = new Target!Parameter;
4                         modelParam.name = parameter.getName();
5                         modelParam.comment = parameter.getDescription();
6                         modelParam.mandatory = parameter.getRequired();
7                         modelParam.type = typeMatch(parameter.getType());
8                         method.params.add(modelParam);
9             }
10      }
```

Listing 5.5: Swagger2Resource for SRM3

SRM4: Body/formData Parameter - Entity Attributes

In Swagger, body or formData parameters (but not both together) can be defined for each operation. They determine the schema of the request body. In case of PUT requests or GET responses, their message structure directly represents the structure of the underlying resource. This is why it semantically fits into the concept of entity structures of the Resource Model. However, this is not necessarily true for POST requests because they are not restricted to enclose the whole resource structure which is why they are processed in a special way, explained in the following.

As illustrated in Listing 5.6, first it must be ensured that only body/formData parameters of POST and PUT methods are processed (Line 1 + 3). Within the operation object, Swagger users typically reference to the definitions section in order to define the schema of an object. Inline definitions are supported but not recommended due to its restricted reusability. Thus, the *defObject* variable passes only the JSON reference, e.g. #/definitions/Pet, to a method called *handleReferences* returning the whole definition object of this pointer (Line 6). Currently, the two predominant data models *ArrayModel* and *RefModel* are supported in the transformation process. Other Swagger data models, e.g. inline definitions, will not be transformed.

```
1       if(parameter.getIn() =="body" or parameter.getIn() =="formData"){
2             // for GET and DELETE no body/formData parameters are allowed
3             if(not (method.eClass().name == "GetMethod" or method.eClass().name
                  =="DeleteMethod")){
4                   // Usually, Swagger definitions are referenced and not defined inline
5                   // Therefore, this method retrieves the definitions object by its reference
6                   var defObject = handleReferences(parameter.getSchema());
7                   if(defObject.isDefined()){
8                         // defObjectGET is the definitions object of a GET method on the same
                              resource
9                         if(method.eClass().name == "PutMethod" and not
                              (defObject.equals(defObjectGET))){
10                              System.err.println("Error occurred. PUT and GET define different
                                    Entity Structures on the same resource");
11                        }else{
12                              // produces an entity object out of the definition object and
                                    attaches it to the resource
13                              createEntityOutOfDefinition(defObject,resource);
14                        }
15                  }
16            }
17      }
```

Listing 5.6: Swagger2Resource for SRM4

If a definitions object could be retrieved, another check ensures the integrity of the Swagger description (Line 9): If the GET method consumes a body parameter, then the PUT method on the identical resource must have defined exactly the same schema as a body parameter. Otherwise an error message will be printed (Line 10). If no error was found, entity attributes will be created for the definitions object with the help of the operation *createEntityOutOfDefinition* expecting the definitions object and the resource which it must be attached to. Again, nested objects cannot be transferred into the Resource Model.

The transformation from entity structures of the the Resource Model back to Swagger parameters is shown in Listing 5.7. If a resource defines entity attributes, a new Swagger definitions model *ModelImpl* will be instantiated (Line 2). After mapping the properties of each entity structure to object properties of Swagger, the Swagger object model will be added to the Swagger model. By default, the definitions object name is always the path name plus the String "_DEF" (Line 11). Finally, a link must be generated in order to reference from the operation's section to the schema definition of the corresponding objects. This is done by the *createLinktoDefinition* ETL method which basically produces body parameters or response definitions for operations of a resource.

```
1       if(not(resource.entityStructure.isEmpty())){
2             var modelImpl = new Target!ModelImpl;
3                   modelImpl.setType("object");
4             for(entityPara in resource.entityStructure){
5                   var property = new Target!ObjectProperty;
6                   // enrich the property variable with the attributes of the entity structure
7                   // [code omitted]
8                   modelImpl.addProperty(property);
9             }
10            var definitionsName = pathName + "_DEF";
11                  createLinktoDefinition(swaggerPath, definitionsName);
12            // add definition section at the end of the transformation process
13            // [code omitted]
14
15      }
```

Listing 5.7: Resource2Swagger for SRM4

SRM5: Path Parameter - Entity Attribute/URL Mapping

Another parameter type is the path parameter in Swagger which is denoted as the dynamic URL fragment in the URL Model. Similar to SRM4, parameters of this type are mapped to entity structures of the Resource Model (SRM5a). Usually, they contain identifying information such as an identification number. Since every operation defines its path parameter, a duplicate detection must avoid defining the same path parameter as entity attributes multiple times.

Actually, the Resource Model does not have a standalone feature to mark entity attributes as path parameters. Whereas static fragments of the URL are directly stored into the URL Model, entity attributes originating from path parameters are referenced from the URL Model using dynamic URL fragments. For this reason, it is possible to identify path parameters within the Resource Model by looking up the URL attributes of the URL Mapping (SRM5b).

According to Listing 5.8, the first step is to check if the given *targetName* (an URL fragment of the Swagger path) contains an opening bracket. If not, a static URL fragment gets directly added to the *Mapping* of the URL Model (Line 27-30). If the fragment indeed contains an opening bracket, this clearly indicates a path parameter because this character is not allowed in a static URL. As discussed before, a duplicate detection must be performed (Line 6-11). Basically, it compares the names of the list of entity attributes for a resource with the current parameter name of the Swagger object. If there already exists an entity structure with the same name for the same resource, an index is returned which is bigger than -1. In this case, a reference to the URL model is established if not already present.

If the current Swagger parameter does not exist in this list (Line 13-19), a new entity attribute is created and enriched with the properties of the Swagger parameter. Finally, the entity structure is linked to the resource by defining it as the *parentResource*. Furthermore, a new dynamic URL fragment is generated for the URL Model, referencing from the URL Model to the previously created entity attribute.

```
1       // targetName is a url fragment
2       if(targetName.contains("{")){
3         for(parameter in parameters){
4             // only path parameters are transformed into entity attributes
5             if(parameter.getIn()=="path"){
6                   var targetRes =Target!Resource.allInstances().selectOne(r| r.name =
                        targetName);
7                   // this variable contains the index of the list of entity structures which
8                   // have the same name as the current "parameter" variable
9                   var indexOfTargetRes =
                        targetRes.entityStructure.name.indexOf(parameter.getName());
10                  // indicates that this parameter was not yet added as an entity structure
11                  if(indexOfTargetRes == -1){
12                  // [code for metamodel cast omitted]
13                        var   entityAtt = new Target!EntityAttribute;
14                        // enrich entity attribute with properties of the Swagger
15                        // parameter [code omitted]
16                        entityAtt.parentResource =
                              Target!Resource.allInstances().selectOne(r|r.name = targetName);
17                        var dynamicURL = new Deploy!DynamicURLFragment;
18                        dynamicURL.attribute = entityAtt;
19                        mapping.url.add(dynamicURL);
20                  }else{
21                        // check if there is already a reference from the URL model
22                        // to this existing entity attribute if not, then create a reference
23                        // [code omitted]
24                  }
25            }
26      }else{
27            var staticURL = new Deploy!StaticURLFragment;
28            var relativePath = targetName.substring(sourceName.length);
29                  staticURL.fragment = relativePath;
30                  mapping.url.add(staticURL);
31      }
```

Listing 5.8: Swagger2Resource for SRM5

Line 20 containing the *else* block is important if an entity attribute has been defined before. By coincidence, the schema of the resource could already contain the path parameter. In this case, the entity attribute was added but there is no reference yet to a dynamic URL fragment. Therefore, the else block generates a new dynamic URL fragment if necessary. The downside of this approach is that path parameters are added as entity attributes even if they do not belong to the schema of a resource as defined by the Swagger description.

In order to explain the discussed issues, see the example Swagger description in Listing 5.9. The path */pets/{petId}* defines the two HTTP methods GET and POST. Both define the path parameter *petId*. Without the described duplicate detection, both parameters would have been added as entity attributes. However, the GET method references to a definitions object called *Pet* which already defined the *petId* property. Since entity attributes are generated before

the path parameters in the transformation process, *petId* was already added. Thus, when processing the first path parameter, only a new dynamic URL fragment must be generated, referencing to the existing *petId* attribute. When processing the second path parameter, both the entity attribute and the URL fragment are already added, so there is nothing more to do.

```
"paths":{
   "/pets/{petId}":{
     "get":{
       "parameters": [
        {
        "name": "petId",
        "in": "path",
        "description": "ID of pet to fetch",
        "required": true,
        "type": "integer"
        }
       ],
       "responses":{
         "200":{
           "schema":{
               "$ref":"#/definitions/Pet"
           }
         }
       }
     },
    "post":{
       "parameters":[
         {
           "name":"petId",
           "in":"path",
           "required":true,
           "type":"integer"
         }
       ],
            //response definition omitted
     }
   }
 },
 "definitions":{
   "Pet":{
     "type":"object",
     "properties":{
       "petId":{"type":"integer"},
       "name":{"type":"string"},
       "tag":{"type":"string"}
     }
   }
 }
```

Listing 5.9: Example Swagger description for the duplicate detection

Some Swagger definitions do not specify an explicit path parameter in the parameter section although their relative path clearly contains at least one. Although this is an obvious fault, the Swagger parser still reads those descriptions, so this issue must be dealt with. Chapter 5.1.2 *Missing Path Parameters* discusses the solving approach.

The transformation back to the Swagger model is illustrated in Listing 5.10. First, all instances of URL references within the URL Model are fetched (Line 1). Then, it is checked for each entity attribute of every resource if the given entity attribute is referenced from the URL model (Line 3). If so, then a new path parameter will be instantiated. After mapping the already discussed properties, the path parameter is added to the list of all parameters within a Swagger path.

```
1       var urlEntityRefs = Deploy!DynamicURLFragment.allInstances().attribute;
2       for(entityPara in resource.entityStructure){
3             if(urlEntityRefs.indexOf(entityPara) > -1){
4                   var pathParam = new Target!PathParameter;
5                         pathParam.setIn("path");
6                         pathParam.setName(entityPara.name);
7                         pathParam.setDescription(entityPara.comment);
8                         pathParam.setType(entityPara.type.name.toLowerCase());
9                         //path parameters MUST be required according to Swagger Spec 2.0
10                        pathParam.setRequired(true);
11                        allParam.add(pathParam);
12                  }
13            }
```

Listing 5.10: Resource2Swagger for SRM5

SRM6: Response - Entity Attributes

The last mapping rule considers response objects of Swagger. In fact, the Resource Model does not have features to define the structure of response messages. However entity structures may carry part of the information. Besides description and response code, the response object of Swagger contains a schema defining the structure of the response body. The response schema can be compared to body parameters for incoming requests as it represents the resource's structure. For this reason, the mapping to entity structures is suited. However, all other information such as the response code in Swagger descriptions get lost.

Though, Swagger supports defining multiple response objects within one operation. For example, an operation could describe the response code 400 (Bad Request) providing a schema of the error message. This schema should not be transformed into entity structures as it does not represent the actual resource. In addition, it could define the response code 200 (OK) defining the schema of the real resource. Listing 5.11 shows how this problem is solved in the Swagger2Resource transformation. First, all response objects are retrieved in the scope of one path. After that, the first response object is assumed to carry the actual schema of the

resource (Line 7). However, in case that two or more response objects are defined, the one with the response code "200" is chosen (Line 11). If there is no "200" response code among these objects, then a random response (the first which is processed) will be taken. Finally, as already illustrated in Listing 5.6, the definition object is retrieved via its JSON reference and an entity structure is generated out of that data schema.

```
1       var responseObjects = path.getResponses().values();
2       var chosenResponse = null;
3       for(response in responseObjects){
4             if(response.getSchema().isDefined()){
5                   if(not chosenResponse.isDefined()){
6                         chosenResponse = response;
7                   }else{
8                         // more than one response schema found. Choose the one with 200 status
9                         if(path.getResponses().get("200").isDefined()){
10                                      chosenResponse = path.getResponses().get("200");
11                        }else{
12                              System.err.println("WARNING: Could not decide which response to
                                     choose. Random response chosen ");
13                        }
14                  }
15            }
16      }
17      if(chosenResponse.isDefined()){
18            var defObject = handleReferences(chosenResponse.getSchema());
19            if(defObject.isDefined()){
20                  createEntityOutOfDefinition(defObject,resource,defObjectMap);
21                  return defObject;
22            }
23      }
```

Listing 5.11: Swagger2Resource for SRM6

The Resource2Swagger transformation defines the ETL operation *createResponse(definitionsName)*, shown in Code Listing 5.12. It is responsible for generating a Swagger response object. The operation creates a Swagger response object (Line 6) and builds a reference to the schema object which was created before[3]. By default, the response code is "200" with a standard description. Response objects with references to schema definitions are defined for all GET and POST methods. DELETE and PUT should not produce a response body according to the HTTP specification. Though, in order to comply with the Swagger Spec 2.0, they must define at least one response object which is handled by the ETL operation *createResponseWithoutReferences()*.

[3]not shown in this extract

```
1
2  operation createResponse(definitionsName){
3
4        if(definitionsName.isDefined()){
5              var responseMap = new Map();
6              var response = new Target!Response;
7              var property = new Target!RefProperty.asDefault(definitionsName);
8                    response.setSchema(property);
9                    response.setDescription("successfull operation. The response definition
                         contains the entity attributes.");
10                   responseMap.put("200",response);
11                   return responseMap;
12       }else{
13             return createResponseWithoutReferences();
14       }
15
16 }
```

Listing 5.12: Resource2Swagger for SRM6

5.1.2 Special Processing

Some mapping rules already indicated several general problems, others require to enrich or pre-check some issues. Therefore, this section provides information about special processing which does not belong to one single mapping rule.

Establishing Links based on the Hierarchical Structure

Swagger does not provide a way to link resources. However, this is a crucial feature of the Resource Model. In order to generate meaningful models from Swagger definitions, the links are generated based on the hierarchical structure of the Swagger paths. Figure 5.2 illustrates a Resource Model instance which was generated out of the Swagger "Petstore" example[4]. At the time of writing, this is the reference example description of Swagger version 2.0.

Basically, the Swagger2Resource transformation gets a list of paths as input and derives links. For example, the graph illustrates the paths */pet* and */pet/{petId}* in the model. Therefore, a relationship is inferred by the ETL transformation and a link is generated. In turn, */pet/{petId}* has also a sub resource because of the existing path name */pet/{petId}/uploadImage*. At first glance, the path */store/order* seems to be a sub resource of */store*. However, as this base resource is not defined in the Swagger document, */store/order* is handled as a root resource, linking to one child resource called */store/order/{orderId}*.

[4] http://petstore.swagger.io/v2/swagger.json

However, the Resource Model requires to specify an HTTP method in order to establish links to a child resource. This is why all links in the graph originated from an arbitrary HTTP method whereas the target is always a resource. For this reason, the user is asked to specify a global priority list of the four available HTTP methods where the links should be generated from. For example, the user's priority list could be {GET, POST, PUT, DELETE} as it is the case in Figure 5.2. If a resource has just one HTTP method, the priority list is irrelevant.

As generating links is an optional feature of the transformation process, the user can choose if the links should be generated automatically. Otherwise, the resources of the Resource Model are not in relation to each other. Consequently, no URL Model will be generated.

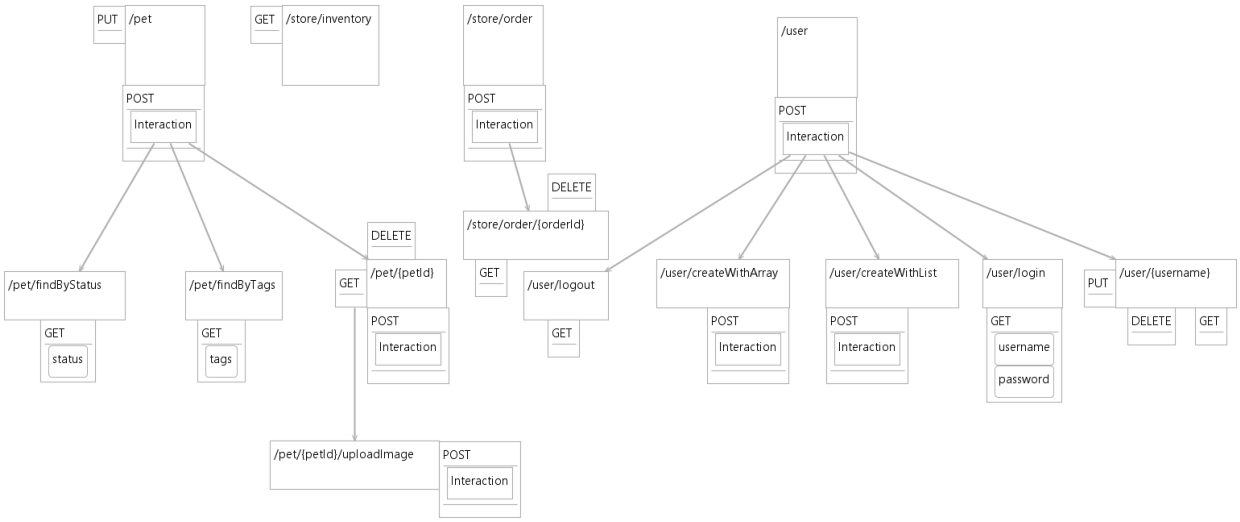

Figure 5.2: Generated Resource Model with automatically created links

Different Entity Structures Defined on the same Resource

There is one conceptual problem when mapping between Swagger parameters and Resource Model's entity structures: While parameters are defined on the operations level, entity structures are attached to resources. As a result, multiple methods could possibly define two different body parameters although they are applied on the same resource. Listing 5.13 shows a conflict between POST and PUT entity structures defined on a resource named *pet*. First, it specifies a POST method in order to add some properties to an existing pet and in addition a PUT method is defined with the purpose of creating a new pet object. Both methods define different body parameter definitions: While the POST method has a special format for just adding new properties to an existing pet, the PUT method requires to contain an entire pet object. In order to resolve this issue, the POST body will only be extracted as an entity structure of the resource if the entity structure could not be established by GET (in the response body) or PUT

(in the request body) methods. The last option is the POST body parameter where a warning will be printed out as this may not be an appropriate behavior for some resources. The reason is that GET and PUT do carry a representation of the resource's structure while POST could define an arbitrary other message format which does not reflect the structure of the resource itself.

```
1     "/pet":{
2        "post":{
3          "summary":"Add new properties to pet object",
4         [code omitted...]
5          "parameters":[
6            {
7              "in":"body",
8              "name":"body",
9              "description":"Pet properties that need to be added to the pet object",
10             "required":true,
11             "schema":{
12               "$ref":"#/definitions/newPetProperties"
13             }
14           }
15         ],
16         "responses":{
17           "405":{
18             "description":"Invalid input"
19           }
20         }
21       },
22       "put":{
23         "summary":"Add a new pet to the store",
24            [code omitted...]
25         "parameters":[
26           {
27             "in":"body",
28             "name":"body",
29             "description":"Pet object that needs to be added to the store",
30             "required":true,
31             "schema":{
32               "$ref":"#/definitions/Pet"
33             }
34           }
35         ],
36         "responses":{
37           "400":{
38             "description":"Invalid ID supplied"
39           }
40         }
41       }
42     }
43   // definitions section omitted
```

Listing 5.13: Swagger description with two different POST and PUT body parameters on the same resource

If the POST method does define a body schema which is different from non-empty PUT or GET entity structures, then the POST structure is transformed into an entity structure of an *interaction* which has been created for this purpose before. This new entity structure strictly relates to the interaction without specifying how the actual resource might look like. This is the way how the description in Listing 5.13 would be transformed.

Nested Objects

The entity attributes of the Resource Model currently do not support nested object definitions. For example, the Swagger definitions section could specify a person object which among others has an address object property. Therefore, the nested object definition (in the example case, the address object) will get lost during the transformation process.

Missing Path Parameters

If a Swagger path contains a path parameter which will be transformed into a dynamic URL fragment the parameter must be defined explicitly with properties such as type, description, name, potential schema, etc. However, some authors of Swagger definitions simply forget specifying path parameters. Though, if no path parameter was specified, there will be no dynamic URL fragment which leads to a semantic error: If the Resource/Model must be transformed back to Swagger description, the transformation assumes that a resource URL containing opening brackets carries a dynamic URL fragment. Since this is not the case if the Swagger parameter was forgotten in the first place, the transformation will produce incorrect descriptions.

In order to solve this issue, Listing 5.14 shows code executed in the end of every Swagger2Resource transformation. Basically, each resource is iterated through, ensuring that each resource name with an opening bracket indeed specifies its dynamic URL fragments. If not, one standard entity attribute and one dynamic URL fragment is created and assigned to the corresponding models. It should be noted that this is adding information which was not there in the Swagger description before. However, it is needed to run the transformations without problems.

```
1       for(resource in allResources){
2             if(resource.name.contains("{")){
3                   var dynFragAll = getAllDynamicFragments(resource.name);
4                   for(dynFrag in dynFragAll){
5                         var existingAtt = resource.entityStructure.selectOne(e|e.name =
                              dynFrag);
6                         if(not existingAtt.isDefined()){
7                               existingAtt = new Target!EntityAttribute;
8                                     existingAtt.name = dynFrag;
9                                     existingAtt.mandatory = true;
10                                    existingAtt.comment = "generated PathParameter";
11                                    existingAtt.type = Target!AttributeDataType#STRING;
12                                    existingAtt.parentResource = resource;
13
14                                    // create Dynamic URL Fragment and reference
15                                    // to this entity attribute. Shown in SRM5 [Code omitted]
16                        }
17                  }
18            }
19      }
20
21 operation getAllDynamicFragments(targetName){
22      var seq = new Sequence();
23      var array = targetName.split("/");
24      for(pathName in array){
25            if(pathName.startsWith("{")){
26                  seq.add(pathName.substring(1,pathName.length()-1));
27            }
28      }
29      return seq;
30 }
```

Listing 5.14: Process missing path parameters

5.2 Model Transformation between RAML and Haupt's Metamodel

In fact, as the metamodel of RAML and Swagger are similar in multiple aspects, their quasi-bidirectional transformation to Haupt's Metamodel does have some commonalities. In order to avoid repeating the same mappings again, this part focuses on documenting just those transformation rules which elementary differ from the mapping rules already described in Chapter 5.1.

Resource (RAML) - Resource (Resource Model)

The equivalent mapping in the Swagger transformation is SRM1. However, there is one fundamental difference between Swagger and RAML resource definitions: While Swagger defines the full relative path on the top level for each resource, RAML uses a hierarchical

structure for their definitions as shown in Example Listing 5.15. However, the Resource Model does not allow nested resources. In order to preserve the resources' hierarchical structure and to avoid the possibility of duplicate resource names, the full relative paths must be transferred into the Resource Model.

```
/user:
      /messages:
            [RAML definition]
            /{messageId}
                  [RAML definition]
      /send:
            [RAML definition]
      /attachments/{id}
            [RAML definition]
```

Listing 5.15: Example of resource definitions in RAML

For this purpose, a recursive ETL operation shown in Listing 5.16 was developed. As input it gets the actual resource of the RAML metamodel, a boolean if it is a root resource, the resource's name and a priority list where the order of the HTTP methods was specified by the user for later link creation.

```
operation createResourceStructure(ramlResource,isRoot, resourceName,priorityList){
      resourceName = resourceName + ramlResource.getRelativeUri();
      if(ramlResource.getActions().isEmpty() and isRoot){
            var keys = ramlResource.getResources().keySet();
            for(key in keys){
                  createResourceStructure(ramlResource.getResource(key), isRoot,
                        resourceName,priorityList);
            }
      }else{
            var targetResource = new Target!Resource;
            targetResource.name = resourceName;
            targetResource.isRoot = isRoot;
            createHTTPMethods(ramlResource.getActions(), targetResource);
            Target!ResourceDiagram.allInstances().first().resources.add(targetResource);
            if(not isRoot and not (priorityList.isEmpty())){
                  createNavigationLink(Target!Resource.allInstances().selectOne(r|r.name==
                        resourceName.substring(0, resourceName.length() -
                        ramlResource.getRelativeUri().length())), targetResource, priorityList);
            }
            isRoot = false;
            if(not ramlResource.getResources().isEmpty()){
                  var keys = ramlResource.getResources().keySet();
                  for(key in keys){
                  createResourceStructure(ramlResource.getResource(key), isRoot,
                        resourceName,priorityList);
                  }
            }
}     }
```

Listing 5.16: RAML2Resource Resource Name Definition

In Line 3, an additional if-conditions takes care of another issue: Given again the example in Listing 5.15, the top level resource */user* is actually not a resource in the sense of the Resource Model because it just has child resources without specifying any own HTTP methods[5]. Thus, if a top level resource just specifies a relative path, the commands in Line 4-6 take care of adding that part of the URL to all child resources without actually creating a new resource. Otherwise (starting from Line 9) a new resource in the Resource Model is created, its RAML actions are handled (Line 12) and then it is added to the *ResourceDiagram*. Afterwards, every child resource gets a *NavigationLink* to its parent (Line 15). Finally, it is checked if the current RAML resource has child resources. If so, these are processed by the very same operation procedure.

The are some other issues within the backwards transformation from the Resource/URL Model to RAML which must be considered. First of all, the full URL paths must be calculated from the Resource and the URL Model. Since this has been already discussed in SRM1, the focus is set on reproducing the hierarchical RAML structure out of the list of complete relative paths. As the source code might be confusing, the following description should provide a basic understanding. The input is the list of complete relative paths.

1) Compare the current resource $\text{url}_{\text{current}}$ with all other paths of the list. If $\text{url}_{\text{current}}$ starts with a compared path which in turn is finalized with a forward slash and both URLs are not equal, mark the compared URL as a potential parent. During further comparisons, in case there is another URL which fulfills these conditions, check if the String length of the current potential parent is smaller. If so, replace the potential parent with the new parent. Proceed with 2a or 2b.

   For example, $\text{url}_{\text{current}}$ is */user/{userId}/messages* which is compared to */user*. As the conditions described are fulfilled, this is the first potential parent. However another comparison to url_{A} */user/{userId}* results in comparing the length of */user* and */user/{userId}*. Since the latter is the longer URL, this is the new potential parent. However, perhaps there is no resource with the URL */user/{userId}*. Then, the direct parent of $\text{url}_{\text{current}}$ is just */user*.

2a) If a parent URL was found, get the actual RAML resources of $\text{url}_{\text{current}}$ and its parent. Calculate the String difference between both URLs which will be the path of the sub resource. Finally, add (or replace if existing) the parent resource with its child resources attached to a final Map which will be used for generating the RAML description.

2b) If a parent URL could not be found, $\text{url}_{\text{current}}$ must be a top level resource. Add it to the final Map.

3) Repeat 1) with the next resource of the list.

[5] called "actions" in RAML

Body Parameters (RAML) - Entity Attributes (Resource Model)

This mapping rule is equivalent to SRM4 in the Swagger transformation. Swagger provides a definitions section within the description for specifying the schema of its parameters which can also be done in RAML. Though, many RAML API developers externalize the schema definition into other files, e.g. written in JSON Schema. It should be noted that the referenced schemas can also be XML Schemas. However, this transformation implementation currently only supports JSON Schema. Anyway, the RAML parser reads external files as plain text. In order to operate on these schemas, an external JSON library called *Jackson* is used as shown Listing 5.17. After building the JSON tree, the JSON Schema should have a JSON node called "properties" (Line 6) specifying the relevant data such as parameter name (Line 11), the mandatory attribute (Line 12) or the type of the parameter (Line 21), while the default type is "object" if no other type was defined. As already described before, a warning message will be shown if the type is not present, an array or an object because schema information will get lost.

```
1  operation createEntityStructure(schema, resource){
2        // the schema is in json string format. Therefore, we need an ObjectMapper
3        // we use the jackson library as a native object to build the json tree
4        var objectMapper = new Native("com.fasterxml.jackson.databind.ObjectMapper");
5        var jsonNode = objectMapper.readTree(schema);
6        var properties = jsonNode.get("properties");
7        var iterator = properties.fieldNames();
8        while(iterator.hasNext()){
9              var element = iterator.next();
10             var entityAtt = new Target!EntityAttribute;
11             entityAtt.name = element.toString();
12             var required = jsonNode.get("properties").get(element).get("required");
13              if(required.isDefined()){
14                   if(required.toString() == "true"){
15                         entityAtt.mandatory = true;
16                   }else{
17                         entityAtt.mandatory = false;
18                   }
19             }
20             var type;
21             if(jsonNode.get("properties").get(element).get("type").isDefined()){
22                   type= jsonNode.get("properties").get(element).get("type").toString();
23             }else{
24                   type = "object"; //default type is object
25             }
26             entityAtt.type = typeMatch(type);
27             if(type.toUpperCase().contains("ARRAY") or type.toUpperCase().contains("OBJECT")
                   ){
28                   System.err.println("Loosing Information due to nested definitions");
29             }
30             entityAtt.parentResource = resource;
31 }     }
```

Listing 5.17: Raml2Resource: Generating entity structures out of plain text schemas

The strategy of transforming entity attributes back to RAML is also slightly different from what was explained in the Swagger transformation. Since every resource has its own entity attributes in the Resource Model, the benefit of sharing definitions cannot be exploited. In the Resource2Swagger transformation, every entity structure has its own schema definition in the definitions section. In contrast, RAML only provides inline definitions or external files. In order to avoid massive generation of external files[6], the inline definition option has been chosen. The implementation was developed according to Listing 5.7.

It should be noted that the JSON Schema does not support descriptions or comments on single parameter values within the "properties" section but only on the object level. This means that comments and descriptions on parameters will get lost during the transformations.

5.3 Conceptual Mapping between RADL and Haupt's Metamodel

The third and final mapping is conducted with RADL and Haupt's Metamodel. Since RADL is rather an academic approach than a widely used REST API DL, the mapping was carried out on a conceptual basis. Unlike for Swagger and RAML, this model transformation has not been implemented. Figure 5.3 illustrates the result of the mapping approach. Since an illustration with arrows would have been too confusing, the mapping is established based on numbers. Model elements tagged with the same number mark semantic correspondence. These elements which do not carry any numbers[7] cannot be represented in the other metamodel.
The Appendix lists a complete example transforming a RADL description file (Listing A.1) into one Resource Model (Listing A.2) and one URL Model (Listing A.3) instance. The following sections explain the rules applied on this transformation.

Resource - Resource (1)

Both metamodels support the concept of resources. In contrast to Swagger or RAML, RADL allows resources to carry a name which is independent of the actual URL path which is identical to the Resource Model. In fact, the semantics of a resource is the same in both metamodels. However, the hierarchical structure and their relations to other elements is different: The central element in the Resource Model is the resource element where links, HTTP methods and entity attributes are attached to. In contrast, the central element in RADL is the HTTP method which has references to almost all other elements. Consequently, a resource in RADL does not carry any information about the entity structure.

[6]For example, a RAML API description with 35 resources could reference a maximum of 35 files
[7]These elements are completely white whereas the others have a gray scale color

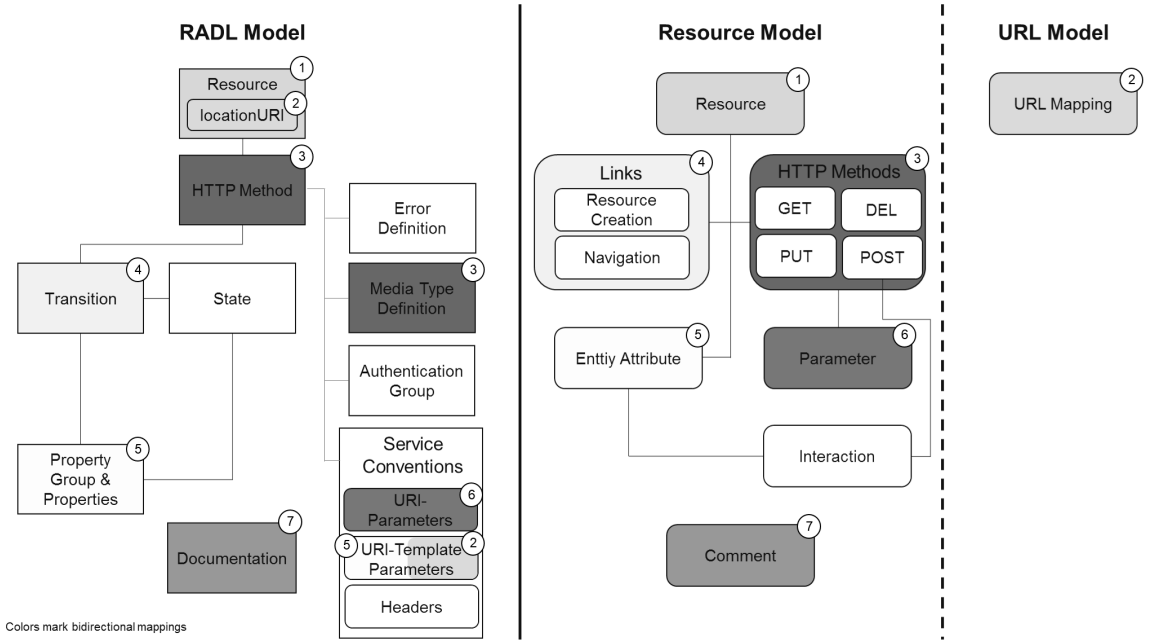

Figure 5.3: Conceptual Mapping between RADL and Haupt's Metamodel

locationURI - URL Mapping (2)

A resource in the RADL model holds a property called *locationURI* representing the actual URL path. This is different to the Resource Model because it does not support such an attribute. Instead, URL information is stored in the URL Model. The *locationURI* in RADL could contain fixed relative paths and URL path parameters. Both can be transformed into corresponding mappings in the URL Model as already discussed in the Swagger transformation.

HTTP Method/Media Type Definition - HTTP Methods (3)

In fact, also the concept of HTTP methods is very similar in both metamodels except for naming conventions. RADL defines *request* and *response* tags to specify the media type of incoming and outgoing messages, while the Resource Model names these attributes *consuming* and *producing*. In contrast to Haupt's Metamodel, the RADL model has a global place for specifying more details of the applied media types (e.g. documentation, specification, media-type-schema, etc.) where the user can also determine a default media type which is applied on all HTTP methods. Whereas the Resource Model makes a distinction about each method, RADL treats all methods the same. Furthermore, both models allow to specify URL parameters for each method. The biggest difference is that methods in RADL specify at least one reference to transitions pointing to states. In contrast, the source of an equivalent link in the Resource Model is indeed a HTTP method, but it targets a resource.

Transition - Link (4)

The mapping between transitions in RADL and links in the Resource Model is complex because transitions connect states, whereas links connect resources. States are abstract constructs which are not supported in the Resource Model. However, they are implemented by the HTTP methods of resources. Therefore, simple relationships between states can be inferred and transferred into the Resource Model, although some constellations are not expressible in the Resource Model as discussed later. Figure 5.4 shows all states and transitions defined in Listing A.2. Figure 5.5 illustrates the resulting Resource Model instance which was manually transformed from the RADL description file.

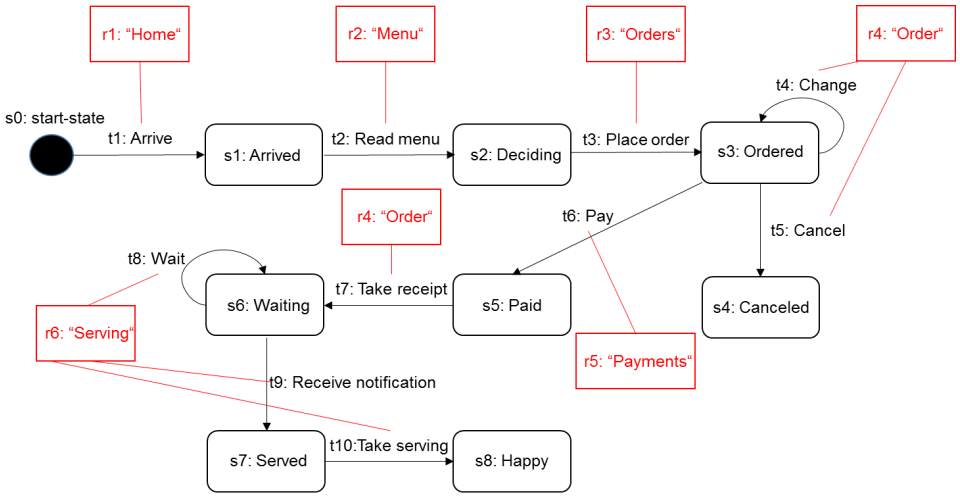

Figure 5.4: The state machine and its implementing resources from Listing A.1

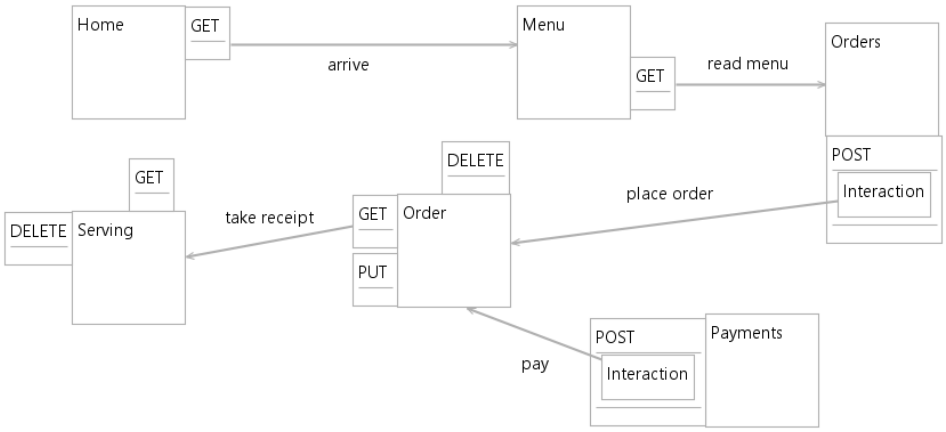

Figure 5.5: The Resource Diagram transformed from Listing A.1

The first three transitions (t1 - t3) can be intuitively transferred because the state machine defines a sequence of states and transitions which are implemented by only one resource which in turn has only one method defined. In the Resource Model, this is represented by a sequence of resources defining links from their single method to the next resource.

The transition "Place orders" (t3) targets the state "Ordered" (s3) which has transitions implemented by the resource "Order" (r4). Therefore, a link from "Orders" (r3) to "Order" (r4) is reasonable in the Resource Model. Though, the state s3 defines transitions named "Change" (t4) and "Cancel" (t5) which are implemented by a PUT and a DELETE method of resource r4. Although the HTTP methods can be modeled in the Resource Model, information will get lost. For example, a DELETE on the resource r4 will abort the hole process which cannot be expressed in the Resource Model.

The transition "Pay" (t6) is also problematic in the transformation process: It is implemented by the resource "Payments" (r5). One might assume a link from r4 to r5. However, r4 does only define three HTTP methods: DELETE points to no resource, PUT points to itself and GET targets only the resource "Serving" (r6). Therefore, there is no way in the Resource Model to capture this. However, t6 points to the state "Paid" (s5). That, in turn, has an outgoing transition called "Take receipt" (t7) which is indeed implemented by r4. Since r5 defines a POST method, a link from r5 to r4 can be established.

The transition t7 points to the state "Waiting" (s6) defining two transitions which are implemented by the resource "Serving" (r6). Therefore, a link from the GET method of r4 to the "Serving" resource is reasonable. The customer has to wait until he receives a notification that he will be served. After he has been served (s7), the serving (and the order) can be deleted after visiting the state "Take serving" (s8). In the Resource Model, the semantics of all these operations will get lost. In the RADL description, the user knows that he has to call the GET method multiple times until he received a notification (t9) that he gets served, otherwise he has to wait (t8) in the same state . Afterwards he knows that he should call the DELETE method.

The same problem is also present for the payment process: In the Resource Model, the user did not get the explicit information (although it can be inferred) from the Resource Model that he should pay (t6) before he can take the receipt (t7). In addition, after he has paid, he cannot call the DELETE method for canceling or the PUT method for changing the order but he can only call the GET method to take the receipt. This information will get lost, too, after the transformation process to the Resource Model.

Property Group & Properties - Entity Attribute (5)

Property groups and entity attributes share the same semantics: They define data schemes for their carrying model elements. While entity attributes are kept simple in this version, property groups typically reference to schemas outside of the RADL description. In case of

the Resource Model, entity attributes can be attached to resources and to interactions. On the contrary, the RADL metamodel allows property groups to be referenced from transitions and states. Usually, states reference property attributes but there is also the possibility to define an *input-property-group* for a transition.

One way to transform property groups into entity attributes is to follow a transition to the destination state which defines a property group. After converting the property groups to entity attributes out of the schemas (simplified), the entity attributes should be attached to the resource implementing the transition. However, this leads to several other problems:

First, if a resource implements several transitions which all reference to multiple different property groups, it is not clear which property group should be transformed. For example, this is the case of the transition "Take receipt" (t7). It has a reference to the property group "receipt" defining several attributes. However, t7 is implemented by a GET method of the resource "Order" (r4) which already has a property group defined by its PUT method: the "order" property group. A solution could be found if t7 was implemented by a POST method since interactions in the Resource Model support another entity structure. Otherwise, one property-group will get lost. This leads directly to the second problem: The *input-property-group* can only be considered if the implementing method is a POST method.

URI-Template Parameters - Entity Attribute/URL Mapping (2 and 5)

URI-Template parameters are handled exactly as Swagger's path parameters. The only difference is that URI-Template parameters are defined globally. For a detailed explanation, look at Chapter 5.1 SRM5.

URI-Parameters - Parameter (6)

URI-Parameters in the RADL model are equivalent to *parameters* in the Resource Model. URI-Parameters could define a separated scheme via property groups, whereas parameters in the Resource Model currently only support some attributes. Furthermore, URI-Parameters are defined globally and can be used in every resource, while parameters in the Resource Model are defined within each HTTP method.

Documentation-Comment (7)

The documentation feature in RADL is the equivalent to the comment attribute in the Resource Model. Almost all model elements contain this feature in both metamodels. One slight difference is that the documentation feature in RADL allows plain text as well as HTML content which can be rendered when generating human-centered documentation.

6 Implementation

This chapter is devoted to explain the more technical aspects of the model transformations' implementation. More precisely, the technical setup describing the transformation architecture and a specific problem during the ETL development called *metamodel conversion* will be addressed.

6.0.1 Technical Setup

First of all, the setup will be explained for conducting the model transformations. Regardless if RAML or Swagger descriptions are involved in the transformation process, the setup does not need to be changed. This is why only one general transformation context will be described fitting for both quasi-bidirectional model transformations.

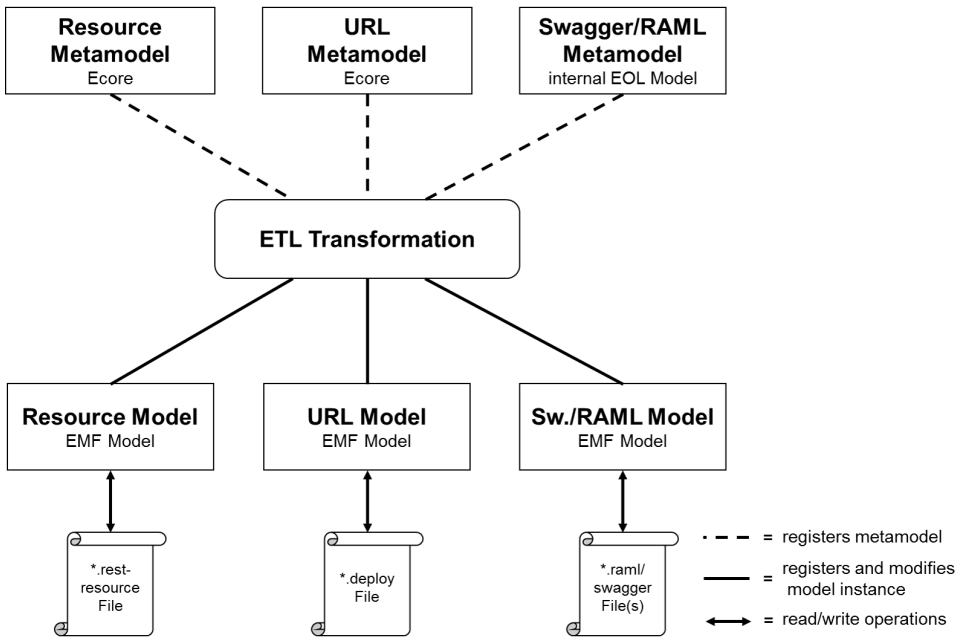

Figure 6.1: The artifacts and their relations within the model transformation

Primarily, all required artifacts must be identified. As shown in Figure 6.1, the central component is the transformation artifact itself written in ETL. In order to modify model instances, the ETL transformation needs the corresponding metamodels to be registered. Only Ecore and EMF Models can be registered to the ETL transformation. The required Resource and URL Model from Haupt's Metamodel are essential which both are originally available as Ecore models.

Additionally, the Swagger/RAML metamodel is required which is not available in Ecore format. Fortunately, both description languages provide open source parsers in Java. These projects contain the languages' metamodel in form of Java classes. With the help of a Java Reflection library, this set of Java classes can be injected into the *JavaModel* class provided by Epsilon (*org.eclipse.epsilon.eol.models.java.JavaModel*). Its purpose is to transform Java classes into an internal Epsilon Object Language (EOL) model representation which is the native language within ETL transformations.

In the same turn of registering the metamodels, the model instances must be added, too. Listing 6.1 shows the Java method called to register all models before executing the ETL transformation. Expected constructor parameters of the *JavaModel* (Line 9) are a unique name, which used within the ETL transformation to refer to the model, a list of Java objects which represent the model instance and finally the set of Java classes, representing the metamodel. The other models (Line 10 + 11) are registered similarly: A unique name is followed by the model instance which is a location path to a serialized model in form of a file. Then, the metamodel is passed in Ecore format. The two boolean parameters determine first if the model should be read on load and second if the models should be stored if an error occurs within the transformation.

```
1  public List<IModel> getModels() throws Exception {
2
3        List<IModel> models = new ArrayList<IModel>();
4
5        // Code omitted for reading all Java classes of Swagger/RAML via reflection
6        // Code omitted for parsing a Swagger/RAML description
7        // and adding the Java object to the "objectList"
8
9        models.add(new JavaModel("SwaggerModel", objectList, classSet));
10       models.add(createEmfModel("ResourceModel", targetDirectory + File.separator
11                       + fileNameResource, "/models/resource.ecore", false, true));
12       models.add(createEmfModel("URLModel", targetDirectory + File.separator
13                       + fileNameDeploy, "/models/deploy.ecore", false, true));
14       return models;
15 }
```

Listing 6.1: Registering metamodels and model instances

One important aspect is that the registration of the models does not specify the roles of source and target models. This is left to the ETL transformation where source and target models are defined due to the *rule* definition. Nevertheless, in the actual implementation code (Listing 6.1 shows just the concept) the models are called "Source" or "Target" in order to avoid vagueness. Obviously, the names change according the transformation. For example, in the Swagger2Resource transformation, the Swagger model is the source model, whereas in the Resource2Swagger transformation, it is the target model.

Since now, the bidirectional nature of the described setup has not been discussed, yet. Figure 6.2 illustrates the flow from one model transforming into another. In general, there can be two directions: top-down or bottom-up. For example, if a Swagger description is given, the file is deserialized via a parser into a Java object. The *JavaModel* creates an EOL Model which can be transformed into an EMF Model of the Resource/URL Model. These final model instances are serialized. In contrast, if a Resource/URL Model is given, there is another ETL transformation producing a Java object since it is not necessary, but possible, to produce an EMF model. Finally, the Java object is serialized into the correct Swagger or RAML format. Summarizing, there is no single bidirectional transformation but two independent unidirectional ones for each transformation process.

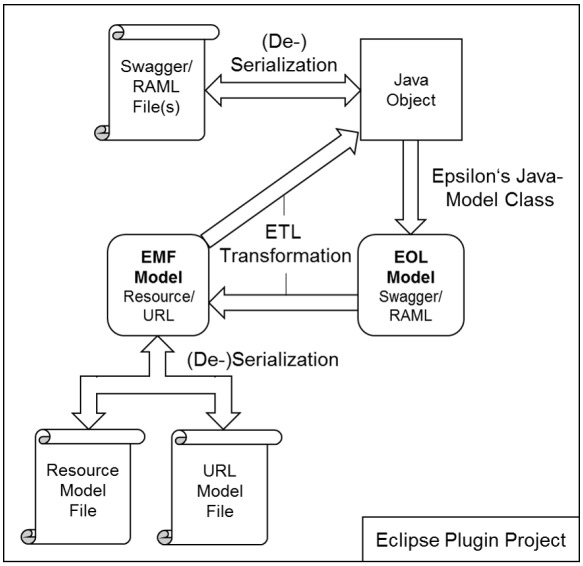

Figure 6.2: Flow of the bidirectional model transformations

Usually, ETL transformations can be started directly by an ETL engine, so no Java is involved. This is convenient if source and target model are available as EMF models. However, since we need to operate with Java objects and Java classes, ETL must be bootstrapped from Java.

Therefore, the *EOL Standalone Example*[1] was taken as a framework to build a customized Java project containing all classes and *.etl files to conduct the transformations.

Going one step further, the Java project was transformed to an Eclipse Plug-in project as Figure 6.2 already indicates. Hence, the usability of performing these transformations could be improved significantly. After starting the Eclipse application, a native context menu is provided for choosing which transformation should be performed. Source file(s) and destination folder can be specified easily. During the transformation process, the user has to make some decisions related to transformation specific issues. Instead of the console, native Eclipse windows will pop up asking for some user input.

6.0.2 Metamodel Conversion

By its Ecore definition, the URL Model references to multiple elements of the Resource Model. For example, the mapping element of the URL Model defines two attributes called *source* and *target* which expect the type *Resource* of the Resource Model. Code extract 6.2 creates a *Mapping* and retrieves the first *Resource* instance of the list of all existing *Resource* instances. Now, a link to that first resource instance should be established in the URL Model by assigning this resource to the source attribute of the *Mapping* variable. Although the procedure seems to be rational, it fails as shown in Listing 6.2.

```
1        var mapping = Deploy!Mapping.createInstance();
2        var resource = Target!Resource.allInstances().first();
3
4        mapping.source = resource;
5
6  // Line 4 will throw this ClassCastException:
7   /*   Internal error: java.lang.ClassCastException: The value of type
         'org.eclipse.emf.ecore.impl.EClassImpl@6f8060ac [name: Resource] [instanceClassName:
         null] [abstract: false, interface: false]' must be of type
         'org.eclipse.emf.ecore.impl.EClassImpl@41b6ae51 [name: Resource] [instanceClassName:
         null] [abstract: false, interface: false]' */
```

Listing 6.2: Metamodel Cast Problem

The reason for this exception is that *mapping.source* expects an input of type *Resource* of the URL Model and not of the Resource Model. Indeed, the URL Model is able to create a *Resource* type instance due to its reference to the corresponding Ecore class of the Resource Model. However, this specific *Resource* type is nothing more than a link to an original resource of the Resource Model. The problem is that there are already existing resources within the Resource Model which we want to reference from the URL Model. Consequently, a new creation is out of question. Fortunately, the class `org.eclipse.emf.ecore.impl.DynamicEObjectImpl` provides

[1] https://www.eclipse.org/epsilon/examples/index.php?example=org.eclipse.epsilon.examples.standalone

a method called *eSetClass(EClass eClass)* with the following description:“Sets the meta class. An object with a set meta class is, by definition, a dynamic instance [...]”[2]. It helps solving this problem as shown in Listing 6.3.

```
1       var mapping = Deploy!Mapping.createInstance();
2       var resource = Target!Resource.allInstances().first();
3       var dummyResource = Target!Resource.createInstance();
4       var deployResource = Deploy!Resource.createInstance();
5       // sets the Metaclass of the resource variable to the Resource type
6       // of the URL Model
7       resource.eSetClass(deployResource.eClass());
8       mapping.source = resource;
9       // set the Metaclass back to original Target!Resource
10      resource.eSetClass(dummyResource.eClass());
11      delete dummyResource;
```

Listing 6.3: Metamodel Cast

In addition to the last version, two new instances must be created: a *dummyResource* variable for carrying the original *Resource* metaclass and a *deployResource* variable for creating an instance of the URL Model *Resource* type. The crucial operation is on Line 7 when the Eclass of the existing *resource* variable is set to the Eclass of the URL Model *Resource* type. After this conversion, the *mapping.source* assignment works. Because ETL uses assignment by reference for native types, the Eclass of the original resource has been modified. Therefore, the Eclass conversion must be set back (Line 10) in order to use the initial resource object during later processing. The same procedure is conducted when referencing entity attributes and links within the URL Model.

[2]http://download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org/eclipse/emf/ecore/impl/DynamicEObjectImpl.html

7 Evaluation of the implemented Model Transformations

This chapter is devoted to evaluate the implemented model transformations. In the context of software, Ross et al. defined evaluation as "*a systematic, rigorous, and meticulous application of scientific methods to assess the design, implementation, improvement, or outcomes of a program.*" [RLF03]. Evaluation approaches can be further distinguished between the following two categories [CRS79]:

Quantitative Evaluation All data collected must be quantifiable. Often, it involves some statistical analysis. The measurements are objective and controlled. Data results must be replicable.

Qualitative Evaluation Data is non-numerical and often involves non structured information such as plain text descriptions. The information content is subjective, often gathered within reviews or sampling.

When deciding which approach to choose, it is recommended to use both methods in order to establish a high confidentiality. Therefore, Chapter 7.1 covers the quantitative approach by applying the transformations on a considerable number of input descriptions. Thereafter, Chapter 7.2 starts with the qualitative evaluation by picking some random samples of the transformations' output files and manually assessing their quality. Errors or warnings are described in text format. Finally, the second part of the qualitative evaluation contains a round trip analysis which will be discussed in Chapter 7.3.

7.1 Applying the Transformations on Real World REST API Descriptions

In order to evaluate the implemented transformations, the first step is to search for real world Swagger and RAML API description files where the transformations can be applied on. Both, RAML and Swagger, advertise on their official websites that powerful companies actively use their description language. For example, Swagger claims that companies such as Paypal, Microsoft, Apigee, Getty Images and others are their customers, while RAML is used by Cisco, Spotify, Oracle's Cloud Service, etc. However, it is hard to find currently active REST API

descriptions in both languages. For instance, the Paypal website obviously uses Swagger tooling for their API documentation, but the formal API description is hidden. In general, it is very cumbersome to find single API definitions.

Fortunately, there are two Github repositories collecting several REST API descriptions. A great number of Swagger definitions are available at `https://github.com/APIs-guru/api-models`. Though, it is a community driven project, meaning that everyone is able to submit an API description, not only the API owners. The acceptance criteria for adding a new API are that it is publicly available, persistent and describes some useful functionality. Obviously, also API definitions originally written in another format can be found as Swagger descriptions. For example, some REST API definitions by Google can be found although they are originally written in a proprietary format called "Google API Discovery Format"[1]. Even though the confidence of these descriptions can be doubted, they are used for evaluation purposes.

In contrast, the repository at `https://github.com/raml-apis` storing RAML definitions, has a public read access but it can only be changed or added by the author. The RAML descriptions in this repository are also referenced from the official documentation website[2]. Consequently, these RAML descriptions should be more reliable.

For conducting the evaluation, the latest version of both repositories are checked out on November 5, 2015. The evaluation process starts with parsing each description file into Java. For Swagger, the official *Swagger Parser*[3] was used, for RAML the corresponding *RAML-Java Parser*[4]. If there are severe parsing errors hindering building the Java objects, the transformations are not initialized. Otherwise, the first ETL transformation transforms the API description into an instance of the Resource/URL Model. After this transformation has been successful, the model instance of the Resource/URL Model is transformed back into the original format.

The purpose of this evaluation approach is to assess if the transformations can be considered *successful* which is the case if the following requirements are met:

1. If a valid input model is given, the transformation runs without throwing exceptions. This is checked by examining the log files and finding the output files which are not generated in case of errors.
2. If a valid input model is given, the transformation should produce a valid output model instance. In case of RAML/Swagger descriptions as output, this is checked by again parsing the output document with the RAML/Swagger parser. In case of a Resource/URL model instance, the two files are opened by the running EMF application for designing new REST services. If the output files can be opened, the model instance passes the validation check.

[1] https://developers.google.com/discovery/v1/reference
[2] http://docs.raml.org/apis
[3] https://github.com/swagger-api/swagger-parser
[4] https://github.com/raml-org/raml-java-parser

Table 7.1: Results of the Automatic Evaluation Run

	Swagger		**RAML**	
	Swagger2 Resource	Resource2 Swagger	Raml2 Resource	Resource2 Raml
Total Number of Input Descriptions/Models	185	90	45	40
Parser Errors	95	0	4	0
Amount of Transformations finished successfully	90	90	40	40

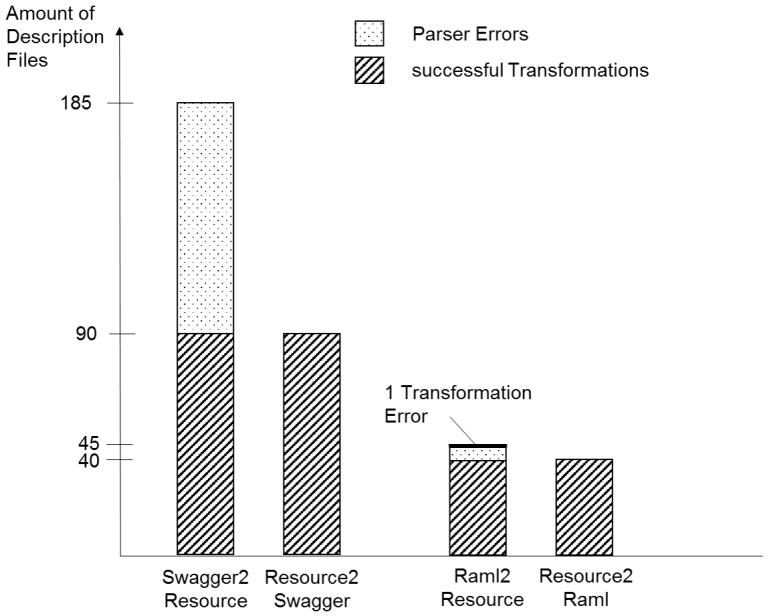

Figure 7.1: Bar chart visualizing the evaluation results of Table 7.1

Table 7.1 and Figure 7.1 illustrate the result of the automated evaluation process. Over 51 % of all tested Swagger descriptions cannot be parsed at all. Many parser errors are caused by the class *PropertyDeserializer* which is responsible for transforming Swagger schemas into Java objects. For instance, errors occur if a property is of type *String* but has the format *int64* which clearly is a mistake by the API designer. One possible reason for this high amount of invalid Swagger descriptions could be that the mentioned Swagger repository has no strict admission criteria regarding the description's validity. However, in case the Swagger description can be parsed, the Swagger2Resource transformation always runs successfully.

While running, the Swagger2Resource transformations printed 544 warnings because information got lost due to nested object definitions. Moreover, 17 resources defined different entities for GET and PUT. The reasons will be discussed later.

Similar results were obtained in the subsequent Resource2Swagger transformations: All 90 models could be transformed back to Swagger definitions successfully. Furthermore, all generated Swagger instances can be parsed by the Swagger Parser, indicating that the instances are valid.

Regarding the RAML transformation evaluation, the descriptions in the collecting repository must be filtered because of the limited support of the ETL transformations regarding schemas: Currently, only JSON Schema is supported. However, 18 RAML descriptions define at least one XML Schema file which is why they have to be sorted out. Finally, 45 RAML descriptions are left referencing just to JSON Schemas. In contrast to Swagger descriptions, the amount of parser errors is minimal. Indeed, the RAML parser could first parse all RAML descriptions without errors. However, exceptions were thrown later because a schema file was missing or the JSON schema could not be parsed. There is also one RAML description causing an internal error within the transformation because one RAML resource does not provide a specific "relativeURI" property. However, as this is a rare case, it was decided to not support such resources.

Overall, 40 valid Resource/URL model instances could be generated. The transformation back to RAML descriptions did not cause any trouble since all created RAML definitions could be parsed successfully.

While executing, the 40 Raml2Resource transformations generated 1582 warnings because of nested objects and 79 failures of the consistency check. In general, referencing different schemas for PUT and GET on the same resource could have the following reasons:

Wrong use of PUT Method The API designer could have used the PUT method in a wrong way. For example, the *NewRelic* RAML description in the mentioned repository defines the path */servers/{server_id}{mediaTypeExtension}*. Its GET method defines the attributes of a server instance such as *id*, *display_name* or *host*. However, the PUT method references a schema which just defines the attribute *display_name*. Because of the PUT method description which is: “This API endpoint allows you to rename your server”, it can be clearly inferred that PUT was used to change an existing attribute. Though, this is a precise action of a POST request, called *partial update*.

Business Reasons for different Entity Structures Sometimes, there are good reasons for different body schemas for incoming and outgoing messages at one resource. For example, a user buys an article in a Web shop. The PUT request includes payment information such as bank account details. For reasons of confidentiality, the GET request could mask some data or leave them out.

Of course, there might also be other reasons for different entity structures for PUT and POST. However, ascertaining the portion of false consistency check warnings is out of scope for this thesis.

7.2 Evaluation of the Transformations' Quality

The previously explained automated evaluation could not assess the quality of the transformation process. The transformation could have left resources out or misses other important aspects while still running without producing failures. Thus, this section reveals the results of manually analyzing and validating ten API descriptions for each description language, taken from the two repositories described in Chapter 7.1. The definitions are chosen randomly with one exception: Files with over 2000 lines of code will not be inspected because of the time needed to manually validate those. In order to pass the validation check, input and output models must have the same amount of resources, the same URL information and common model elements such as HTTP methods, schema information, query/path parameters, etc. must be reflected in both models equally. Information should only get lost if the counterpart does not support storing this kind of data. Finally, the description is searched for unexpected or suspicious constructs or model elements.

Table 7.2: Results of the Manual Validation with Swagger Descriptions

API Description Name	Swagger2Resource	Resource2Swagger
bikewise.org	✓	✓
bitdango.com	✓	✓
cambase.io	∼	✓
citycontext.com	∼	✓
clarify.io	∼	✓
cybertaxonomy.eu	✓	✓
datumbox.eu	✓	✓
eriomem.net	✓	✓
firebrowse.org	✗	✓
geneea.com	✓	✓

✓ = nothing found ✗ = Error ∼ = Warning

Table 7.2 shows the result of the manual validation conducted by the author of this thesis. Check marks indicate that no defective or abnormal behavior could be detected. Tildes mean that there is a minor issue, similar to a warning message. Finally, the cross symbol reveals that a crucial error was found. The corresponding error or warning descriptions can be found subsequently.

firebrowse.org (Error) The resource structure contains the paths */Metadata/SampleTypes* and */Metadata/SampleType/Code/{code}*. These resources must have a link which actually is not present, though. Actually this is an error of the implementation code.

cambase.io (Warning) This REST API Description uses media type extensions within the URL, for example */api/v1/models.json* or */api/v1/models/search.json*. Actually, a link must be established between these paths. However, this is not recognized by the transformation due to their media type extensions. Though, REST API best practices recommend to put the media type into header fields. Furthermore, this API description heavily relies on formData parameters which are not supported by the Swagger transformation. The Swagger file contains 1855 lines of code, whereas the Resource Model instance only contains 88 code lines, so a lot of information got lost.

citycontext.com (Warning) One of the Swagger paths is named */@{lat},{lon}*. In fact, this path contains two path parameters for *latitude* and *longitude*. This is not recognized by the Swagger transformation because parameters do not have a leading forward slash.

clarify.io (Warning) The definitions model heavily uses so called *RefModels*. These are not supported. Therefore, most schema information gets lost. Excluding this issue, the rest of the transformation works fine.

The same procedure was conducted with RAML descriptions, too. The results are illustrated in Table 7.3. A general issue occurs in Resource2Raml descriptions when a resource defines a media type which is not *application/json*. As the transformation currently only supports JSON Schema, the schema will always be defined in JSON even if XML is the media type. Nevertheless, this is not seen as a major problem, since the transformation could be extended with XML Schema support.

Table 7.3: Results of the Manual Validation with RAML Descriptions

API Description Name	Raml2Resource	Resource2Raml
accuweather.com	∼	✓
blockscore.com	∼	✓
ewaypayments.com	✓	✓
freebase.com	✓	✓
googleapis.com/calendar	✓	✓
googleapis.com/gmail	✓	✓
outlook.office365.com	✓	✓
paypal.com	✓	✓
uber.com	✓	✓
wordpress.com	✓	✓

✓ = nothing found ✗ = Error ∼ = Warning

accuweather.com (Warning) A similar issue occurs as mentioned before in the *cambase.io* Swagger description: The RAML file contains paths where media type extensions are explicitly defined in the URL path, this time as path parameters. Consequently, the relative path */locations/version/adminareas/*{*countryCode*}{*mediaTypeExtension*} should have a navigation link to */locations/version/adminareas/*{*countryCode*}*/search*{*mediaTypeExtension*} which is not the case. In addition, the mentioned base resource suddenly has two path parameters in one URL fragment, namely *countryCode* and *mediaTypeExtension* which is also not supported by the transformation.

blockscore.com (Warning) The description provides a response body for a POST request and a different body schema for the GET request of the same resource. The Resource Model has the construct of *inputEntityStructures* which are applied only on POST input body messages, though. Consequently, the GET body schema will be transformed into the entity structure of the resource while the POST response body schema will get lost.

7.3 Round Trip Analysis

A round trip is the process of transforming a REST API description into a Resource/URL Model instance and afterwards transforming the very same output back into the original API definition language. Of course, some information will get lost due to the diversity of the underlying metamodels. As an example, the previously mentioned Swagger petstore example is used for showing how the transformation affects the information content of the descriptions after the transformation round trip. Figure 7.2 illustrates the two high level JSON trees of the original Swagger file and the generated Swagger description after the transformations. Except security definitions and some other general information, the REST API descriptions are quite similar. Amount and name of the Swagger paths did not change but the definitions section did in both respects. Instead of six schema definitions, the generated Swagger specifies eleven ones. In addition, the definition names changed. The reason is that the Resource Model does not support global definitions of entity structures. They must be directly attached to resources. As a consequence, the generated Swagger file specifies for each resource, defining a body or response schema, a separated definitions object.

Since analyzing each resource would go beyond the scope of this example, Figure 7.3 picks one of the Swagger paths, namely */pet*, comparing the original and the generated Swagger paths. As one can see, the HTTP methods PUT and POST are defined in both Swagger descriptions. Though, some general information get lost such as *operationId* or *summary*. The *description* field would have been obtained, but in this case, it is empty. Incoming and outgoing media types are equal in both Swagger paths. The body parameters of both HTTP methods are retained although name and description dropped out. Since the Resource Model does not support response codes, all response information get lost except of a default response schema. Security definition are not tracked at all.

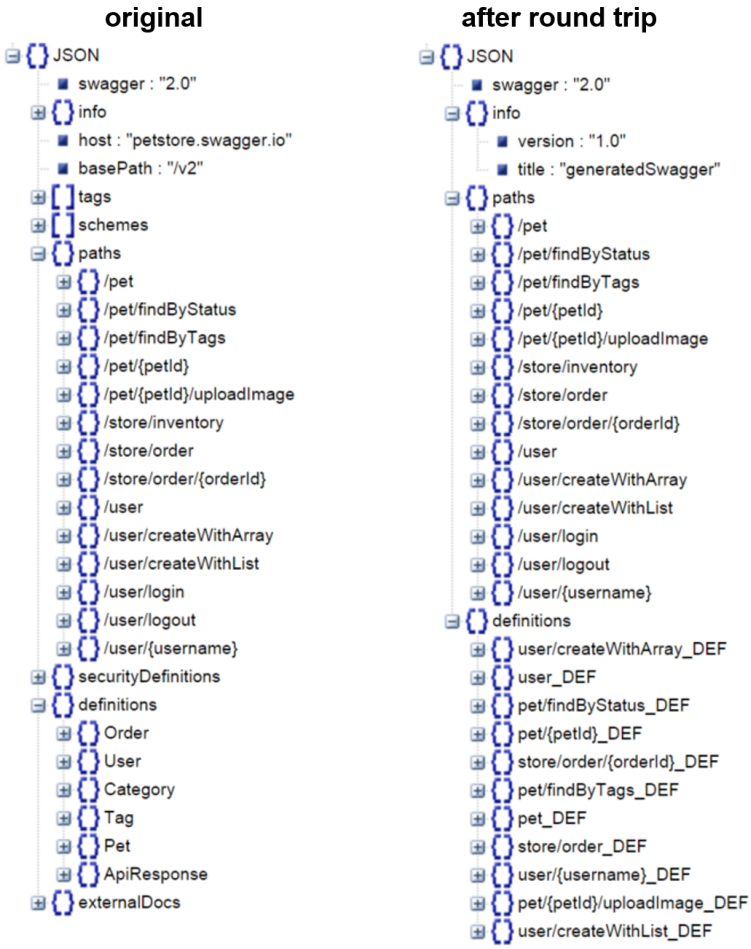

Figure 7.2: High level JSON tree comparison of the original and generated Swagger file

Finally, 7.4 shows the definition objects which are referenced by the previously discussed Swagger paths. It is easy to see that the generated definition lacks nested object definitions. For example, the attribute *category* actually contains a reference to an object definition called *Category*. After the round trip, the generated definition only knows that the *category* property is an object type.

If the generated Swagger file of the petstore example runs through a second round trip, the swagger description stays the same. However, this may not be true for all API descriptions. In general, the original description is far smaller by size as the transformed one. The reason is that global schema definitions are copied multiple times as the Resource Model does not support them. Especially, this effect was noticeable in RAML where reuse of schema definitions seems to be very popular.

Figure 7.3: Comparison of the path */pet*

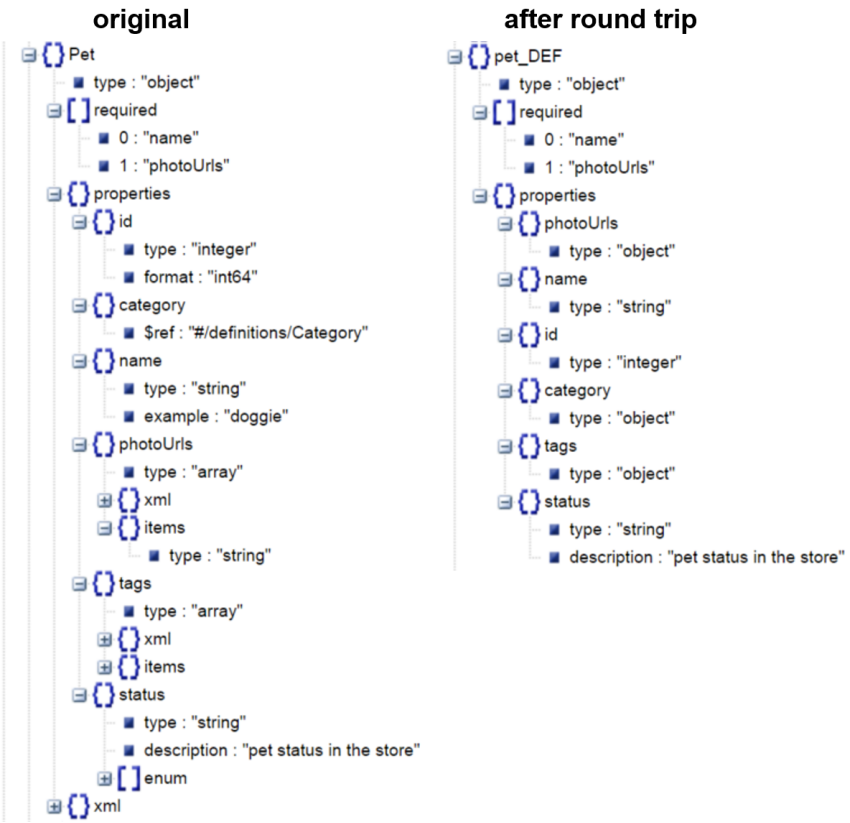

Figure 7.4: Comparison of the schema definitions

8 Discussion

Actually, many elements of the Swagger and RAML metamodels could be transformed into corresponding model elements of Haupt's Metamodel and vice versa as demonstrated in the transformation mappings in Chapter 5. The fact that the transformations could produce meaningful description models in both directions shows that all description languages examined do follow a common approach. However, there are also some essential differences between them. The following sections highlight the benefits of one metamodel compared to the other. Since RAML and Swagger share a similar metamodel they are grouped as one in most respects.

Advantages of Haupt's Metamodel over Swagger/RAML

The following list discusses the advantages of Haupt's Metamodel compared to Swagger and RAML.

Separation of Resource Definition and URL Information In contrast to the Resource Model, Swagger & RAML do not have independent resource names which also represent the fixed URL paths. The Resource Model separates URL information from the resource definition. The Swagger/RAML clients are tightly coupled as they rely on a predefined URL structure. Dynamic changes or creation of new resources leading to new URLs is not representable in the Swagger/RAML models.

Links between Resources and HATEOAS Swagger & RAML do not support HATEOAS. Consequently, links between resources cannot be defined. Strictly speaking, the industry API DLs cannot be used to design a truly RESTful API. The Github Issue Number 97[1] in Swagger's official Github repository, discussing hypermedia support for Swagger, was closed as "*not per design*". This statement was confirmed by Reverb's Tony Tam, one of the driving forces behind Swagger, during an interview in June 2015 [Amu15a]. He argues that HATEOAS support does not contribute to any of the current Swagger goals. Furthermore he is not sure if hypermedia should be supported at all in Swagger descriptions.

[1] https://github.com/swagger-api/swagger-core/issues/97#event-159248864

Likewise, RAML does not provide accurate support for HATEOAS. Uri Sarid, one of the RAML creators, states in an interview in April 2015 [Amu15b] that people already describe hypermedia APIs with RAML although it is not supported natively. He infers that the designers use a special format such as the Hypertext Application Language (HAL) throughout the definition of parameters and responses. Still, resource relations cannot be defined within the RAML description. Sarid mentions that suggestions already have been made to integrate hypermedia support for RAML. Though, they have "*not found a way to do so in a format agnostic way*" [Amu15b].

Supporting REST and HTTP compliance In fact, many Swagger and RAML descriptions examined in Chapter 7 violated against REST constraints and HTTP standards. Besides the critical HATEOAS violations, Swagger and RAML descriptions also misused the uniform interface constraint, for example using PUT for partial updates as demonstrated in Chapter 7.1 or defining schemas of input/output messages which should not be defined according to the HTTP specification. Haupt's Metamodel avoids these problems by following various approaches: First, the Resource Model could be generated from a REST-independent Domain Model. The transformation engine automatically generates Resource Models using the uniform interfaces correctly. Second, when using composite Resource Models, this building set already guarantees the right use of the uniform interfaces. Finally, the Resource's Metamodel already restricts some anti patterns as their HTTP methods adhere to the HTTP specification. Defining different entity structures for PUT and GET methods is not even possible.

Visualization RAML and Swagger require the user to actually describe the REST API in a structured format such as YAML or JSON. While their design editors give the users a supporting integrated development environment (IDE), they still have to write source code. This will require REST API experts, otherwise design errors are likely, for example, forgetting path parameters as discussed in Chapter 5.1.2. On the contrary, the Resource Model is developed by drawing visual model elements. Human beings typically prefer visualization over textual formats.

Model-driven Approach As discussed in Chapter 2.3, the model-driven nature provides several benefits such as usability improvements, avoiding repetitive tasks, fewer errors, better portability and maintainability, increasing productivity, etc.

Advantages of Swagger/RAML over Haupt's Metamodel

However, there are also crucial advantages of the Swagger/RAML metamodel over the Resource/URL Model. We do not consider the tooling around the description languages.

Definition of Data Schemas One very obvious problem is the simplified schema definition of the Resource Model. The evaluation showed that most data got lost due to nested object definitions in Swagger/RAML descriptions. In contrast, the industry REST API DLs

support a rich schema definition, for example a subset of the JSON Schema (Swagger) or full schema definitions in JSON, YAML, XML and others (RAML).

Reuse of global Data Schemas Another crucial benefit of RAML and Swagger is that they allow to reuse schema definitions. They both define sections to define data schemas which can be referenced from multiple locations. This enhances maintainability while reducing code size and complexity of the descriptions.

Path Parameters In Swagger and RAML, path parameters can be defined explicitly as one specific type in the parameters section of one HTTP method. In the Resource Model, the only way of defining path parameters is by adding an entity structure attribute to the resource and linking this specific attribute from the URL model. This has several drawbacks: First, there is a semantic problem since path parameters are not always part of the actual resource‘s structure. Second, base resources, which are not targets of a link, cannot carry path parameters at all. However, defining path parameters in the Swagger/RAML way has the disadvantage of defining the same path parameter for every HTTP method of a resource.

Inheritance (only RAML) RAML offers the possibility to define resource types and traits. In fact, these are mother classes of resources and methods. The concept is very convenient if some resource and/or various HTTP methods share common properties. This approach improves maintainability and reduces code size, thus improving readability of the descriptions by introducing one level of abstraction. Haupt's Metamodel does not provide inheritance features but Haupt introduced a higher level of abstraction by the use of *conversations* which basically are building blocks for modeling high-level capabilities [HLP15].

Design Flexibility Swagger and RAML basically give the users plenty of rope for designing their own customized REST API. On the one hand, users could design poor API descriptions violating against best practices and REST constraints. On the other hand, this freedom can be exploited to describe customized interfaces which sometimes have special needs. For example, defining different data schemas for GET and PUT on the same resource which is not possible in the Resource Model. Anyway, experts are required to design high-quality descriptions.

Extended Features Swagger and RAML provide ways to capture general information such as version, the purpose of the API, etc. which is not possible in the Resource Model. Furthermore, the industry REST API DLs provide features for including example requests and responses, links to external documentation, extensibility features for customized definition fields or a dedicated support to specify security definitions. All these features rather aim to support the user in practical concerns than introducing new conceptual ideas.

Comparison of the RADL approach with Haupt's Metamodel

The biggest advantage of the RADL approach compared to the Resource/URL Model is clearly that it supports the definition of states and transitions. This abstract construct allows to separate implementation from concept and therefore enhances reusability. Another consequence is that the RADL description contains more semantics such as the right sequence of actions which should be performed on a resource. In addition, RADL also supports *link-relations* in order to specify the semantics of transitions. Furthermore, RADL supports reusability of implementation constructs due to the global definition of media-types, errors and other *conventions* which can be used within the resources.

Besides the already mentioned benefits of Haupt's Metamodel, one crucial disadvantage of RADL is its complexity and the resulting design errors. The RADL description file references a lot of other model elements, so its dependency is high while reducing its clarity. This negative effect is reinforced by designing RADL descriptions by hand. Another problem is that multiple different property-groups could be defined on one resource as it was the case for the transition "take receipt" in Figure 5.4. Actually the resource contains two or more independent entity structures which is bad practice. Finally, RADL does not separate resource descriptions from URL information.

Discussing Adaptations to the Resource Model

The Resource Model needs a more sophisticated approach of defining data schemas. Of course, one can propose to include support for XML or JSON Schema. However, this would highly increase the complexity of defining entity structures. A trade-off would be to just allow nested object definitions. By estimation, this would cover most object definitions in the examined REST API descriptions. However, there should also be the possibility to reference to other objects. Proposed is an approach which is used in Swagger and RAML: Defining global data schemas which can then be referenced from a resource by pointing to a specific object definition.

Furthermore, the following solution is proposed in order to overcome the problem of path parameters: As it is already the case of query parameters, a new model element called *path parameter* should be defined in the Resource Model. However, in contrast to attach path parameters to an HTTP method, they are linked to the actual resource. This provides the advantage of not defining path parameters for every implemented HTTP method as it is required in Swagger and RAML. Since path parameters are no objects but primitive types (mostly *String* or numbers), a flat schema definition should be enough including name, description and type. The *mandatory* field can be left out because path parameters are always required as part of the URL.

Moreover, it would be convenient to define some global information such as version, license, contact information, external links, etc. In addition, globally defining default media types for incoming and outgoing messages would reduce the need to repeat the media types for HTTP methods which do not differ from the default media type definition.

In contrast to the Resource Model, response definitions in Swagger and RAML include a response code and a description of the response code (besides the response message body schema). However, it is controversial if this is really an advantage. For example, the RADL authors argue that response code documentation encourages the client to just expect the described ones. If the server changes, the client will break. Therefore, it is better that clients should have general status code handling capabilities [RSW15]. This infers to use default response codes whose semantic has already been described instead of customized response code descriptions.

The RAML concept of inheritance seems to be useful for textual descriptions. It forces the user to think of abstract types before modeling. This could result in higher quality descriptions. However, it must be evaluated if this approach could also be useful in a model-driven context with the Resource Model.

Finally, the RADL idea of defining states and transitions in combination with typical resource descriptions, which actually implement them, is promising. Furthermore, it would seamlessly fit into the model-driven approach since modeling states and transitions can be perfectly represented by state-machine diagrams. However, the basic architecture of the Resource Model must be changed in order to support this concept. Again, a further analysis is needed in order to decide if this change is reasonable.

9 Conclusion

The main goal of this thesis was to compare the academic Metamodel by Haupt et al., especially the Resource and the URL Model, with already existing approaches for describing REST APIs which are widely used in industry today. After identifying Swagger and RAML as leading REST API DLs on the market, quasi-bidirectional model transformations were conceptually designed between each of the two industry REST API DLs and Haupt's Metamodel. Therefore, the semantics of the model elements had to be analyzed. If matching elements of different metamodels were found, the reasons for the corresponding mappings were discussed. In a second step, the model transformations were implemented using the transformation language ETL after the transformation architecture had been set up. Then, a third conceptual mapping between another research-oriented description language called RADL and Haupt's Metamodel was established but not implemented. However, the implemented model transformations were evaluated with quantitative and qualitative methods. The evaluation shows that all implemented model transformations can be applied on various real-world REST API descriptions, producing accurate results in most cases. The quality of the transformation was evaluated by manually reviewing random output models comparing them with the original source models. Except for some special cases, the output models met the expectations. Finally, strengths and weaknesses of each metamodel were reflected, always in comparison to Haupt's Metamodel. Derived from the results of this thesis, propositions were made in order to improve and extend the existing metamodel.

In summary, it can be stated that Haupt's Metamodel indeed already defines several core elements of the industry REST API DLs as the model transformations showed. However, there are also some crucial differences. Arguably, the biggest one is the support of the essential REST concept HATEOAS. Whereas Haupt's Metamodel provides the possibility of linking resources while separating resource descriptions from URL information, Swagger and RAML provide none of these features. Consequently, both industry description languages cannot be used to describe hypermedia APIs according to Richardson's maturity level 3. Nevertheless, these languages are widely used since most existing "REST" APIs in industry are, in fact, violating against HATEOAS. On the contrary, the model transformations showed among other issues that the Resource Model's way of defining entity attributes is too simplified, compared to the rich support of defining JSON or XML Schema in Swagger/RAML. Eventually, the description language RADL indeed supports HATEOAS by defining states and transitions in addition to the usual resource descriptions. Thus, it is possible to define more semantics into the descriptions compared to

the Resource/URL Model. But RADL's complexity is high and it is much more difficult to write accurate descriptions, compared to the model-driven nature of Haupt's Metamodel.

In the future, the proposed enhancements to the Resource Model should be evaluated. If the integration into Haupt's Metamodel seems appropriate, the next step would be extending the Resource/URL Model with the new features which were derived from other REST API description languages while still preserving the simplicity of designing REST APIs.

A Appendix

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
## Sample RADL file for RESTBucks.
## Copyright (c) EMC Corporation. All rights reserved.
-->
<service name="RESTBucks" radl-version="1.0"
   xmlns="urn:radl:service" xmlns:radl="urn:radl:service"
   xmlns:html="http://www.w3.org/1999/xhtml">
 <documentation>
   This example service follows <html:a
       href="http://www.infoq.com/articles/webber-rest-workflow">RESTBucks</html:a>,
   an online version of coffee shop Starbucks based on Gregor Hohpe's
   <html:a
       href="http://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html">observation</html:a>
       that
   it is an asynchronous processing pipeline.
 </documentation>
 <states>
   <start-state>
    <transitions>
     <transition name="Arrive" to="Arrived">
      <documentation>The process starts when the customer walks into the
          store.</documentation>
     </transition>
    </transitions>
   </start-state>
   <state name="Arrived" property-group="home">
    <transitions>
     <transition name="Read menu" to="Deciding">
      <documentation>
       Once the customer has arrived in the store, she reads the menu to see what the store
           offers.
      </documentation>
     </transition>
    </transitions>
   </state>
   <state name="Deciding" property-group="menu">
    <transitions>
     <transition name="Place order" to="Ordered">
      <documentation>
       The customer composes an order from the items on the menu and places the order with
           the cashier.
```

```
37          </documentation>
38          <input property-group="order"/>
39        </transition>
40      </transitions>
41    </state>
42    <state name="Ordered" property-group="order">
43      <transitions>
44        <transition name="Change" to="Ordered">
45          <documentation>As long as the customer hasn't paid, she can change her
                order.</documentation>
46          <input property-group="order"/>
47        </transition>
48        <transition name="Cancel" to="Canceled">
49          <documentation>
50            The customer may decide she doesn't want anything after all and cancel the whole
                 order.
51          </documentation>
52        </transition>
53        <transition name="Pay" to="Paid">
54          <documentation>The customer pays for the order.</documentation>
55          <input property-group="payment"/>
56        </transition>
57      </transitions>
58    </state>
59    <state name="Canceled"/>
60    <state name="Paid" property-group="receipt">
61      <transitions>
62        <transition name="Take receipt" to="Waiting">
63          <documentation>The customer accepts the confirmation of her payment.</documentation>
64        </transition>
65      </transitions>
66    </state>
67    <state name="Waiting" property-group="serving">
68      <transitions>
69        <transition name="Wait" to="Waiting">
70          <documentation>The customer has to wait a while.</documentation>
71        </transition>
72        <transition name="Receive notification" to="Served">
73          <documentation>The barista notifies the customer once he has prepared her
                serving.</documentation>
74        </transition>
75      </transitions>
76    </state>
77    <state name="Served" property-group="serving">
78      <transitions>
79        <transition name="Take serving" to="Happy">
80          <documentation>The customer picks up her serving.</documentation>
81        </transition>
82      </transitions>
83    </state>
84    <state name="Happy"/>
```

```
  </states>
  <link-relations>
    <link-relation name="self">
      <specification href="http://tools.ietf.org/html/rfc4287"/>
      <transitions>
        <transition ref="Receive notification"/>
        <transition ref="Wait"/>
      </transitions>
    </link-relation>
    <link-relation name="https://schema.org/ReplaceAction">
      <specification href="https://schema.org/ReplaceAction"/>
      <transitions>
        <transition ref="Change"/>
      </transitions>
    </link-relation>
    <link-relation name="https://schema.org/DeleteAction">
      <specification href="https://schema.org/DeleteAction"/>
      <transitions>
        <transition ref="Cancel"/>
      </transitions>
    </link-relation>
    <link-relation name="http://schema.org/ReceiveAction">
      <specification href="http://schema.org/ReceiveAction"/>
      <transitions>
        <transition ref="Take serving"/>
      </transitions>
    </link-relation>
    <link-relation name="https://schema.org/PayAction">
      <specification href="https://schema.org/PayAction"/>
      <transitions>
        <transition ref="Pay"/>
      </transitions>
    </link-relation>
    <link-relation name="http://schema.org/menu">
      <specification href="http://schema.org/menu"/>
      <transitions>
        <transition ref="Read menu"/>
      </transitions>
    </link-relation>
    <link-relation name="http://schema.org/OrderAction">
      <specification href="http://schema.org/OrderAction"/>
      <transitions>
        <transition ref="Place order"/>
      </transitions>
    </link-relation>
    <link-relation name="http://schema.org/Order">
      <specification href="http://schema.org/Order"/>
      <transitions>
        <transition ref="Take receipt"/>
      </transitions>
    </link-relation>
```

```
136   </link-relations>
137   <property-groups>
138     <property-group name="home" uri="http://schema.org/CafeOrCoffeeShop"/>
139     <property-group name="item" uri="http://schema.org/Product">
140       <property name="name" uri="http://schema.org/name"/>
141       <property name="size" uri="http://schema.org/height"/>
142       <property name="milk" uri="http://www.productontology.org/doc/Milk"/>
143       <property name="price" uri="http://schema.org/price" type="number"/>
144       <property name="currency" uri="http://schema.org/priceCurrency"/>
145     </property-group>
146     <property-group name="menu" uri="http://schema.org/menu">
147       <property-group name="item" ref="item" repeats="true"/>
148     </property-group>
149     <property-group name="order" uri="http://schema.org/Order">
150       <property-group name="item" ref="item" repeats="true"/>
151       <property name="customer" uri="http://schema.org/customer"/>
152       <property name="total" uri="http://schema.org/totalPrice" type="number"/>
153       <property name="currency" uri="http://schema.org/priceCurrency"/>
154     </property-group>
155     <property-group name="receipt" uri="http://schema.org/Order">
156       <property name="dateTime" uri="http://schema.org/orderDate" type="xsd:dateTime"/>
157       <property name="shop" uri="http://schema.org/seller"/>
158       <property-group name="item" ref="item" repeats="true"/>
159       <property name="total" uri="http://schema.org/totalPrice" type="number"/>
160       <property name="currency" uri="http://schema.org/priceCurrency"/>
161       <property name="paymentMethod" uri="http://schema.org/acceptedPaymentMethod"/>
162     </property-group>
163     <property-group name="payment" uri="http://reference.data.gov.uk/def/payment#payment">
164       <property name="amount" uri="http://schema.org/totalPrice" type="number"/>
165       <property name="currency" uri="http://schema.org/priceCurrency"/>
166       <property name="paymentMethod" uri="http://schema.org/acceptedPaymentMethod"/>
167       <property name="cardholderName" uri="https://w3id.org/creditcard/v1/name"/>
168       <property name="cardNumber" uri="https://w3id.org/creditcard/v1/number"/>
169       <property name="expiryMonth" uri="https://w3id.org/creditcard/v1/expiryMonth"
              type="number"/>
170       <property name="expiryYear" uri="https://w3id.org/creditcard/v1/expiryYear"
              type="number"/>
171       <property name="cardSecurityCode" uri="https://w3id.org/creditcard/v1/verificationCode"/>
172     </property-group>
173     <property-group name="serving" uri="http://schema.org/Order"/>
174   </property-groups>
175   <media-types default="application/ld+json">
176     <media-type name="application/ld+json">
177       <specification href="http://www.w3.org/TR/json-ld/"/>
178     </media-type>
179   </media-types>
180   <errors>
181     <error name="http://errors.restbucks.com/missing-item" status-code="400">
182       <documentation>The order you provided doesn't contain any menu item.</documentation>
183     </error>
184     <error name="http://errors.restbucks.com/unknown-item" status-code="400">
```

```
185       <documentation>The menu item you requested is unknown to the server.</documentation>
186     </error>
187     <error name="http://errors.restbucks.com/invalid-item" status-code="400">
188       <documentation>The attributes you provided are invalid for the requested menu
              item.</documentation>
189     </error>
190     <error name="http://errors.restbucks.com/item-out-of-stock" status-code="400">
191       <documentation>The menu item you requested is temporarily out of stock.</documentation>
192     </error>
193     <error name="http://errors.restbucks.com/missing-customer" status-code="400">
194       <documentation>The order you provided doesn't contain a customer name.</documentation>
195     </error>
196     <error name="http://errors.restbucks.com/invalid-payment" status-code="400">
197       <documentation>The payment details you provided contain invalid values.</documentation>
198     </error>
199     <error name="http://errors.restbucks.com/overpaid" status-code="400">
200       <documentation>The amount of money you paid is more than what the order
              costs.</documentation>
201     </error>
202     <error name="http://errors.restbucks.com/payment-not-processed" status-code="400">
203       <documentation>The payment you provided could not be processed.</documentation>
204     </error>
205     <error name="http://errors.restbucks.com/already-paid" status-code="400">
206       <documentation>The order is already paid and cannot be changed anymore.</documentation>
207     </error>
208     <error name="http://errors.restbucks.com/not-found" status-code="404">
209       <documentation>The URI you requested doesn't exists or is not accessible by
              you.</documentation>
210     </error>
211     <error name="http://errors.restbucks.com/method-not-allowed" status-code="405">
212       <documentation>The method you used is not supported on this URI.</documentation>
213     </error>
214     <error name="http://errors.restbucks.com/not-acceptible" status-code="406">
215       <documentation>
216         The media type you requested in the Accept header is not supported for this method on
              this URI.
217       </documentation>
218     </error>
219     <error name="http://errors.restbucks.com/server-error" status-code="500">
220       <documentation>
221         Something went wrong on our side. We have logged the problem so our staff can look into
              it. We are sorry for the
222         inconvenience.
223       </documentation>
224     </error>
225     <error name="http://errors.restbucks.com/service-unavailable" status-code="503">
226       <documentation>The server is temporarily not able to handle the request.</documentation>
227     </error>
228   </errors>
229   <conventions>
230     <uri-template-variables>
```

```
231       <uri-template-variable name="order-id">
232         <documentation>
233           A unique server-generated ID for an order. The client should treat this as an opaque
                identifier.
234         </documentation>
235       </uri-template-variable>
236     </uri-template-variables>
237   </conventions>
238   <resources>
239     <resource name="Home">
240       <location uri="/"/>
241       <methods>
242         <method name="GET">
243           <transitions>
244             <transition ref="Arrive"/>
245           </transitions>
246           <response/>
247         </method>
248       </methods>
249     </resource>
250     <resource name="Menu">
251       <location uri="/menu/"/>
252       <methods>
253         <method name="GET">
254           <transitions>
255             <transition ref="Read menu"/>
256           </transitions>
257           <response/>
258         </method>
259       </methods>
260     </resource>
261     <resource name="Orders">
262       <location uri="/orders/"/>
263       <methods>
264         <method name="POST">
265           <transitions>
266             <transition ref="Place order"/>
267           </transitions>
268           <request/>
269           <response/>
270         </method>
271       </methods>
272     </resource>
273     <resource name="Order">
274       <location uri-template="/orders/{order-id}/"/>
275       <methods>
276         <method name="DELETE">
277           <transitions>
278             <transition ref="Cancel"/>
279           </transitions>
280         </method>
```

```
      <method name="PUT">
        <transitions>
          <transition ref="Change"/>
        </transitions>
        <request/>
        <response/>
      </method>
      <method name="GET">
        <transitions>
          <transition ref="Take receipt"/>
        </transitions>
        <response/>
      </method>
    </methods>
  </resource>
  <resource name="Payments">
    <location uri-template="/orders/{order-id}/payments/"/>
    <methods>
      <method name="POST">
        <transitions>
          <transition ref="Pay"/>
        </transitions>
        <request/>
        <response/>
      </method>
    </methods>
  </resource>
  <resource name="Serving">
    <location uri-template="/orders/{order-id}/serving/"/>
    <methods>
      <method name="GET">
        <transitions>
          <transition ref="Wait"/>
          <transition ref="Receive notification"/>
        </transitions>
        <response/>
      </method>
      <method name="DELETE">
        <transitions>
          <transition ref="Take serving"/>
        </transitions>
      </method>
    </methods>
  </resource>
 </resources>
</service>
```

Listing A.1: Restbucks Example Description in RADL. Taken from [RSW15]

```
<?xml version="1.0" encoding="UTF-8"?>
<com.rmt.models:ResourceDiagram xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:com.rmt.models="http://rest-modeling/resource/1.0">
  <resources name="Home" isRoot="true">
    <GET>
      <produces>APPLICATION_JSON</produces>
    </GET>
    <entityStructure name="item"/>
    <entityStructure name="name" type="STRING"/>
    <entityStructure name="size" type="STRING"/>
    <entityStructure name="milk" type="STRING"/>
    <entityStructure name="price" type="INTEGER"/>
    <entityStructure name="currency" type="STRING"/>
    <entityStructure name="home"/>
  </resources>
  <resources name="Menu">
    <GET>
      <produces>APPLICATION_JSON</produces>
    </GET>
    <entityStructure name="menu"/>
    <entityStructure name="item"/>
  </resources>
  <resources name="Orders">
    <POST>
      <interactions comment="" name="Interaction">
        <produces>APPLICATION_JSON</produces>
        <consumes>APPLICATION_JSON</consumes>
        <inputEntityStructure name="order"/>
        <inputEntityStructure name="item"/>
        <inputEntityStructure name="customer" type="STRING"/>
        <inputEntityStructure name="total" type="INTEGER"/>
        <inputEntityStructure name="currency" type="STRING"/>
      </interactions>
    </POST>
  </resources>
  <resources name="Order">
    <GET>
      <produces>APPLICATION_JSON</produces>
    </GET>
    <PUT>
      <produces>APPLICATION_JSON</produces>
    </PUT>
    <DELETE>
      <produces>APPLICATION_JSON</produces>
    </DELETE>
    <entityStructure name="order"/>
    <entityStructure name="item"/>
    <entityStructure name="customer" type="STRING"/>
    <entityStructure name="total" type="INTEGER"/>
    <entityStructure name="currency" type="STRING"/>
```

```
50      <entityStructure name="order-id" type="INTEGER" identifier="true"/>
51    </resources>
52    <resources name="Payments">
53      <POST>
54        <interactions name="Interaction">
55          <produces>APPLICATION_JSON</produces>
56          <consumes>APPLICATION_JSON</consumes>
57        </interactions>
58      </POST>
59      <entityStructure name="payment"/>
60      <entityStructure name="amount" type="INTEGER"/>
61      <entityStructure name="currency" type="STRING"/>
62      <entityStructure name="paymentMethod" type="STRING"/>
63      <entityStructure name="cardholderName" type="STRING"/>
64      <entityStructure name="cardNumber"/>
65      <entityStructure name="expiryMonth" type="INTEGER"/>
66      <entityStructure name="expiryYear" type="INTEGER"/>
67      <entityStructure name="cardSecurityCode" type="STRING"/>
68    </resources>
69    <resources name="Serving">
70      <GET>
71        <produces>APPLICATION_JSON</produces>
72      </GET>
73      <DELETE>
74        <produces>APPLICATION_JSON</produces>
75      </DELETE>
76    </resources>
77    <links xsi:type="com.rmt.models:Navigation" relType="Arrive" source="//@resources.0/@GET"
          target="//@resources.1"/>
78    <links xsi:type="com.rmt.models:Navigation" relType="Read Menu" source="//@resources.1/@GET"
          target="//@resources.2"/>
79    <links xsi:type="com.rmt.models:Navigation" relType="Place order"
          source="//@resources.2/@POST/@interactions.0" target="//@resources.3"/>
80    <links xsi:type="com.rmt.models:Navigation" relType="pay" source="//@resources.3/@GET"
          target="//@resources.4"/>
81    <links xsi:type="com.rmt.models:Navigation" relType="take receipt and wait"
          source="//@resources.3/@GET" target="//@resources.5"/>
82  </com.rmt.models:ResourceDiagram>
```

Listing A.2: The manually transformed Resource Model from Listing A.1

```
1   <?xml version="1.0" encoding="UTF-8"?>
2   <com.rmt.models:DeploymentModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:com.rmt.models="http://rest-modeling/deploy/1.0">
3     <mappings information="">
4       <source href="default.restresource#//@resources.0"/>
5       <target href="default.restresource#//@resources.1"/>
6       <url xsi:type="com.rmt.models:StaticURLFragment" fragment="/"/>
7     </mappings>
8     <mappings information="">
9       <source href="default.restresource#//@resources.1"/>
```

```
    <target href="default.restresource#//@resources.2"/>
    <url xsi:type="com.rmt.models:StaticURLFragment" fragment="/menu"/>
  </mappings>
  <mappings>
    <source href="default.restresource#//@resources.1"/>
    <target href="default.restresource#//@resources.2"/>
    <url xsi:type="com.rmt.models:StaticURLFragment" fragment="/orders"/>
  </mappings>
  <mappings>
    <source href="default.restresource#//@resources.2"/>
    <target href="default.restresource#//@resources.3"/>
    <url xsi:type="com.rmt.models:DynamicURLFragment">
      <attribute href="default.restresource#//@resources.3/@entityStructure.5"/>
    </url>
  </mappings>
  <mappings>
    <source href="default.restresource#//@resources.4"/>
    <target href="default.restresource#//@resources.3"/>
    <url xsi:type="com.rmt.models:StaticURLFragment" fragment="/orders"/>
    <url xsi:type="com.rmt.models:DynamicURLFragment">
      <attribute href="default.restresource#//@resources.3/@entityStructure.5"/>
    </url>
  </mappings>
  <mappings>
    <source href="default.restresource#//@resources.3"/>
    <target href="default.restresource#//@resources.5"/>
    <url xsi:type="com.rmt.models:StaticURLFragment" fragment="/serving"/>
  </mappings>
</com.rmt.models:DeploymentModel>
```

Listing A.3: The corresponding URL Model

```
{
  "swagger": "2.0",
  "info": {
    "title": "Swagger Petstore",
    "description": "A sample API to demonstrate features in the swagger-2.0 specification",
    "termsOfService": "http://swagger.io/terms/"
  },
  "host": "petstore.swagger.io",
  "basePath": "/api",
  "consumes": [
    "application/json"
  ],
  "produces": [
    "application/json"
  ],
  "paths": {
    "/pets": {
      "get": {
        "description": "Returns all pets from the system that the user has access to",
        "responses": {
          "200": {
            "description": "A list of pets.",
            "schema": {
              "type": "array",
              "items": {
                "$ref": "#/definitions/Pet"
              }
            }
          }
        }
      }
    }
  },
  "definitions": {
    "Pet": {
      "type": "object",
      "required": [
        "id",
        "name"
      ],
      "properties": {
        "id": {
          "type": "integer",
          "format": "int64"
        },
        "name": {
          "type": "string"
        }
      }
    }
  }
}
```

Listing A.4: Shortened Example of a Swagger Description. Taken from the Swagger repository [Swa15]

Bibliography

[Amu15a] M. Amundsen. APIs with Swagger : An Interview with Reverb's Tony Tam, 2015. URL http://www.infoq.com/articles/swagger-interview-tony-tam. (Cited on page 93)

[Amu15b] M. Amundsen. RAML Founder Talks About the API Industry: Governance, Technology, and Acquisitions, 2015. URL http://www.infoq.com/news/2015/04/raml-interview-uri-sarid. (Cited on pages 44 and 94)

[ASJH11] P. Adamczyk, P. H. Smith, R. E. Johnson, M. Hafiz. REST and Web Services: In Theory and in Practice. In *REST: From Research to Practice*, pp. 35–57. Springer, 2011. (Cited on page 1)

[Bie10] M. Biehl. Literature Study on Model Transformations. Technical Report ISRN/KTH/MMK/R-10/07-SE, Royal Institute of Technology, 2010. (Cited on pages 27 and 28)

[BPR13] M. Bravo, J. Pascual, J. Rodríguez. Semantic Representation of Public Web Service Descriptions. In *ICCSA (5)'13*, pp. 636–651. 2013. (Cited on page 5)

[BS11] N. Bhuvaneswari, S. Sujatha. *Integrating Soa and Web Services*. River Publishers series in information science and technology. River Publishers, 2011. (Cited on page 3)

[CRS79] T. Cook, C. Reichardt, E. R. Society. *Qualitative and quantitative methods in evaluation research*. Sage research progress series in evaluation. Sage Publications, 1979. (Cited on page 83)

[DGD06] D. Djuric, D. Gasevic, V. Devedzic. The Tao of Modeling Spaces. *Journal of Object Technology*, 5(8):125–147, 2006. (Cited on pages 26 and 27)

[Ecl05] Eclipse. The Eclipse Modeling Framework (EMF) Overview, 2005. URL http://help.eclipse.org/juno/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html. (Cited on page 23)

[Ecl07] Eclipse Documentation. Web services overview, 2007. URL http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.jst.ws.doc.user%2Fconcepts%2Fcws.html. (Cited on page 4)

[Ecl15] Eclipse. The Eclipse Modeling Framework (EMF), 2015. URL http://www.eclipse.org/modeling/emf/. (Cited on page 22)

[Erl07] T. Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007. (Cited on page 5)

[Fie00] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, 2000. (Cited on page 8)

[Fie08a] R. T. Fielding. On software architecture, 2008. URL http://roy.gbiv.com/untangled/2008/on-software-architecture. [Online; accessed 12-August-2015]. (Cited on page 8)

[Fie08b] R. T. Fielding. REST APIs must be hypertext-driven, 2008. URL http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven. [Online; accessed 12-August-2015]. (Cited on page 15)

[FR14a] R. Fielding, J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, 2014. (Cited on page 11)

[FR14b] R. Fielding, J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, 2014. (Cited on page 12)

[FR14c] R. Fielding, J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, RFC Editor, https://tools.ietf.org/html/rfc7231, 2014. (Cited on page 34)

[GDD+04] S. Graham, G. Daniels, D. Davis, Y. Nakamura, S. Simeonov, P. Brittenham, P. Fremantle, D. Koenig, C. Zentner. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI*. Pearson Education, 2004. (Cited on page 13)

[GMB09] T. Gherbi, D. Meslati, I. Borne. MDE between Promises and Challenges. In *UKSim*, pp. 152–155. IEEE Computer Society, 2009. (Cited on page 25)

[Hap14] L. Happe. *Configurable Software Performance Completions through Higher-Order Model Transformations:*. The Karlsruhe Series on Software Design and Quality. KIT Scientific Publishing, 2014. (Cited on page 21)

[Her14] L. Heritage. API Description Languages, 2014. URL http://de.slideshare.net/SOA_Software/api-description-languages. (Cited on page 39)

[Heu07] R. Heutschi. *Serviceorientierte Architektur: Architekturprinzipien und Umsetzung in die Praxis*. Business Engineering. Springer Berlin Heidelberg, 2007. (Cited on page 13)

[HFK+14] F. Haupt, M. Fischer, D. Karastoyanova, F. Leymann, K. Vukojevic-Haupt. Service Composition for REST. In *2014 IEEE 18th International Enterprise Distributed Object Computing Conference, EDOC 2014*, pp. 110–119. IEEE, 2014. (Cited on page 11)

[HKLS14] F. Haupt, D. Karastoyanova, F. Leymann, B. Schroth. A Model-Driven Approach for REST Compliant Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2014)*, pp. 129 – 136. IEEE, 2014. (Cited on pages 1 and 31)

[HLP15] F. Haupt, F. Leymann, C. Pautasso. A conversation based approach for modeling REST APIs. In *12th Working IEEE / IFIP Conference on Software Architecture - WICSA 2015*. IEEE Computer Society, 2015. (Cited on pages VI, 31, 32 and 95)

[HT06] B. Hailpern, P. Tarr. Model-driven Development: The Good, the Bad, and the Ugly. *IBM Syst. J.*, pp. 451–461, 2006. (Cited on page 20)

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev. ATL: A Model Transformation Tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008. (Cited on pages VI, 26 and 28)

[KAG15] A. Kriouile, N. Addamssiri, T. Gadi. An MDA Method for Automatic Transformation of Models from CIM to PIM. *American Journal of Software Engineering and Applications Vol. 4, No 1*, pp. 1–14, 2015. (Cited on page 25)

[KRGDP15] D. Kolovos, L. Rose, A. García-Domínguez, R. Paige. The Epsilon Book. Technical report, Eclipse, 2015. (Cited on pages VI and 29)

[Lan14] K. Lane. RAML Specification 1.0, 2014. URL http://apievangelist.com/2014/01/31/the-vision-behind-swagger-api-blueprint-and-raml/. (Cited on page 43)

[LZG04] Y. Lin, J. Zhang, J. Gray. Model comparison: A key challenge for transformation testing and version control in model driven software development. In *Control in Model Driven Software Development. OOPSLA/GPCE: Best Practices for Model-Driven Software Development*, pp. 219–236. Springer, 2004. (Cited on page 1)

[ML09] G. Mak, J. Long. *Spring Enterprise Recipes: A Problem-Solution Approach*. Expert's voice in open source. Apress, 2009. (Cited on page 19)

[MMG+07] J.-J. Moreau, N. Mendelsohn, M. Gudgin, M. Hadley, H. F. Nielsen, Y. Lafon, A. Karmarkar. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C recommendation, W3C, 2007. Http://www.w3.org/TR/2007/REC-soap12-part1-20070427/. (Cited on pages 6 and 7)

[New02] E. Newcomer. *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. Independent technology guides. Addison-Wesley, 2002. (Cited on page 5)

[NLW81] J. Nestor, D. A. Lamb, W. A. Wulf. IDL-interface description language : formal description, revision 10. Technical Report CMU-CS-81-139, Carnegie-Mellon University.Computer science. Pittsburgh (PA US), 1981. (Cited on page 13)

[OAS06] OASIS. *Reference Model for Service Oriented Architecture 1.0*. OASIS, 2006. (Cited on page 3)

[OMG03] OMG. MDA Guide Version 1.0.1, 2003. URL http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf. (Cited on pages VI, 21 and 24)

[OMG06] OMG. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006. URL http://www.omg.org/cgi-bin/doc?formal/2006-01-01. (Cited on page 22)

[OMG15] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.2, 2015. URL http://www.omg.org/spec/QVT/1.2/. (Cited on page 28)

[OMN+04] D. Orchard, F. McCabe, E. Newcomer, H. Haas, C. Ferris, D. Booth, M. Champion. Web Services Architecture. W3C note, W3C, 2004. Http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/. (Cited on page 4)

[Pap08] M. Papazoglou. *Web Services: Principles and Technology*. Pearson education. Pearson Prentice Hall, 2008. (Cited on page 4)

[PER13] A. Poutsma, R. Evans, T. A. Rabbo. Spring Web Services: Reference Documentation. Technical report, Spring Framework, 2013. URL http://docs.spring.io/spring-ws/site/reference/pdf/spring-ws-reference.pdf. (Cited on page 19)

[PP13] L. Panziera, F. D. Paoli. A framework for self-descriptive RESTful services. In *WWW (Companion Volume)*, pp. 1407–1414. International World Wide Web Conferences Steering Committee / ACM, 2013. (Cited on page 16)

[PTDL07] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. 40(11):38–45, 2007. (Cited on page 3)

[PZL08] C. Pautasso, O. Zimmermann, F. Leymann. Restful Web Services vs. "Big"' Web Services: Making the Right Architectural Decision. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pp. 805–814. ACM, New York, NY, USA, 2008. (Cited on pages 7 and 11)

[Ram14] R. Ramanathan. *Handbook of Research on Architectural Trends in Service-Driven Computing*. Advances in Systems Analysis, Software Engineering, and High Performance Computing:. IGI Global, 2014. (Cited on page 15)

[RAM15a] RAML. RAML Specification 1.0, 2015. URL http://docs.raml.org/specs/1.0/. (Cited on pages 43 and 45)

[RAM15b] RAML. RAML Website, 2015. URL http://raml.org/. (Cited on pages 15 and 43)

[RAR13] L. Richardson, M. Amundsen, S. Ruby. *RESTful Web APIs*. O'Reilly Media, 2013. (Cited on pages 9 and 10)

[RCSW13] J. Robie, R. Cavicchio, R. Sinnema, E. Wilde. RESTful Service Description Language (RSDL): Describing RESTful Services Without Tight Coupling. *Presented at Balisage: The Markup Conference 2013, Montreal, Canada, August 6-9*, 2013., 2013. (Cited on page 15)

[Rec08] J. Rech. *Model-Driven Software Development: Integrating Quality Assurance: Integrating Quality Assurance*. Premier Reference Source. Information Science Reference, 2008. (Cited on page 20)

[RGO12] A. Rotem-Gal-Oz. *SOA Patterns*. Running Series. Manning, 2012. (Cited on page 3)

[RLF03] P. Rossi, M. Lipsey, H. Freeman. *Evaluation: A Systematic Approach*. SAGE Publications, 2003. (Cited on page 83)

[Rod15] A. Rodriguez. RESTful Web services: The basics, 2015. URL http://www.ibm.com/developerworks/library/ws-restful/ws-restful-pdf.pdf. (Cited on page 1)

[RSK12] D. Renzel, P. Schlebusch, R. Klamma. Today's Top "RESTful" Services and Why They Are Not RESTful. In *WISE*, volume 7651 of *Lecture Notes in Computer Science*, pp. 354–367. Springer, 2012. (Cited on page 1)

[RSW15] J. Robie, R. Sinnema, E. Wilde. RADL: A description language and tooling for hypermedia-driven RESTful APIs, 2015. URL https://github.com/restful-api-description-language/RADL. (Cited on pages 46, 97 and 107)

[SA11] T. Steiner, J. Algermissen. Fulfilling the Hypermedia Constraint via HTTP OPTIONS, the HTTP Vocabulary in RDF, and Link Headers. In *Proceedings of the Second International Workshop on RESTful Design*, WS-REST '11, pp. 11–14. ACM, New York, NY, USA, 2011. (Cited on page 15)

[SAN15] K. SANDOVAL. Top Specification Formats For REST APIs, 2015. URL http://nordicapis.com/top-specification-formats-for-rest-apis/. (Cited on page 39)

[SB05] A. Staikopoulos, B. Bordbar. Bridging Technical Spaces With A Metamodel Refinement Approach. A BPEL To PN Case Study Abstract, 2005. (Cited on page 28)

[SBPM09] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. (Cited on pages VI, 23 and 24)

[SC13] M. Stephan, J. R. Cordy. Application of Model Comparison Techniques to Model Transformation Testing. In *MODELSWARD 2013 - Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development, Barcelona, Spain, 19 - 21 February, 2013*, pp. 307–311. 2013. (Cited on page 1)

[Sin13] R. Sinnema. Restful API Description Language (RADL), 2013. URL http://www.xmlamsterdam.com/pdf/2013/2013-remonsinnema-radl-xmlamsterdam2013/09-radl.html. (Cited on page 46)

[SR03] W. Sadiq, F. Racca. *Business Services Orchestration: The Hypertier of Information Technology*. Business Services Orchestration: The Hypertier of Information Technology. Cambridge University Press, 2003. (Cited on page 13)

[SRC14] G. C. Silva, L. M. Rose, R. Calinescu. A Qualitative Study of Model Transformation Development Approaches: Supporting Novice Developers. In *Proceedings of the 1st International Workshop on Model-Driven Development Processes and Practices co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), Valencia, Spain, September 28, 2014.*, pp. 18–27. 2014. (Cited on page 2)

[Sta73] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973. (Cited on page 21)

[Sto14] M. Stowe. RAML vs. Swagger vs. API Blueprint, 2014. URL http://www.mikestowe.com/2014/07/raml-vs-swagger-vs-api-blueprint.php. (Cited on page 39)

[SVC06] T. Stahl, M. Voelter, K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. (Cited on pages 20, 21 and 26)

[Swa14] Swagger. Swagger Specification 2.0, 2014. URL http://swagger.io/specification/. (Cited on page 41)

[Swa15] Swagger. Wiki for the Swagger Specification 2.0, 2015. URL https://github.com/swagger-api/swagger-spec/wiki. (Cited on pages 40 and 111)

[VB15] B. Varanasi, S. Belida. *Spring REST*. Apress, 2015. (Cited on page 15)

[VHM+14] R. Verborgh, A. Harth, M. Maleshkova, S. Stadtmüller, T. Steiner, M. Taheriyan, R. Van de Walle. Survey of Semantic Description of REST APIs. In C. Pautasso, E. Wilde, R. Alarcón, editors, *REST: Advanced Research Topics and Practical Applications*, pp. 69–89. Springer, 2014. URL http://link.springer.com/chapter/10.1007/978-1-4614-9299-3_5. (Cited on page 16)

[VSVD+12] R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, J. G. Vallés, R. Van de Walle. Functional Descriptions As the Bridge Between Hypermedia APIs and the Semantic Web. In *Proceedings of the Third International Workshop on RESTful Design*, WS-REST '12, pp. 33–40. ACM, New York, NY, USA, 2012. (Cited on page 15)

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. (Cited on pages 3 and 14)

All links were last followed on January 8, 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature