Towards Interoperability to the Implementation of RESTful Web Services: A Model Driven Approach

Nírondes Tavares and Samyr Vale
Department of Computer Science
Federal University of Maranhão
São Luis – MA – Brazil
nirondes@gmail.com, samyr@deinf.ufma.br

***Abstract* — Nowadays, a issue that has been gaining relevance is the RESTful Web services technology. RESTful has been more prominent than SOAP due to the interoperability gained by the Web support. Nevertheless, the implementation tier of the RESTful Web Services can be developed with various programming languages and with different specifications, frameworks, and plugins. This diversity of ways to implement RESTful Web services reduces the interoperability proposed by this technology and prevents reuse. In this paper, we present a model-driven approach to implement RESTful Web Services. By the use of concerns separation, modeling on different abstraction levels and by the use of transformation rules we solve the interoperability lack.**

Keywords-Web service; RESTful; modeling; transformation; MDE;

I. INTRODUCTION

The development of the Web has transformed the exchange of organizational information. The information that was previously accessed and presented only via browser could, with the advent of Web services, also be accessed by other ways. This allows that besides humans, computer programs could also make use of this information [1].

Among the existing Web service architectures, the Representational State Transfer (REST) [2] has been gaining space especially for the lightweight and convenience in the use and dissemination of information.

RESTful Web services aims to be interoperable, because a client, that uses a specific platform, can establish communication with a server that uses a different platform. But, the interoperability feature is just at the communication level, because both parts communicate through the Hypertext Transfer Protocol (HTTP) layer, which is the standard for the Internet communication. However, RESTful Web services, as well as SOAP ones, need to be implemented with a specific programming language.

Today, there exist numerous of programming languages (e.g., Java, C++, C #, etc) that support the development of such services, and within these languages, there is still a significant amount of different specifications, Application Programming Interfaces (APIs), frameworks, and plugins. Face of this variety of ways, the problem of interoperability arises in the implementation tier of RESTful Web services. To be interoperable, the programming languages must conform to a certain degree of compatibility with the others [3].

Model Driven Development has been applied in different issues as an approach which provides, by concerns separation, the development of business logic independent of technologies and programming languages. Model driven approaches, such as Model Driven Engineering (MDE) and Model Driven Architecture (MDA) provide interoperability by metamodeling and transformation techniques.

MDE uses the model as the main artifact and the transformation as the principal activity in system modeling and development.

In [4], we have presented the use of MDE to the development of the syntactic and the semantic description of Semantic RESTful Web Services. In this paper, we propose a MDE-based approach to solve the problem of the lack of interoperability in the Application tier of the RESTful Web Services technology. Thus, we present the WSSR metamodel proposed for the implementation of interoperable RESTful Web Services. We also present a target platform metamodel and the transformation rules to the implementation of the RESTful Web Service in the chosen target language.

This paper is organized as follows. Section 2 presents an overview of the main technologies used in the proposed solution. Section 3 presents our model-driven approach for implementing RESTful Web services. Section 4 presents a case study with a sample implementation of a RESTful Web service. Finally, Section 5 concludes the paper.

II. TECHNOLOGICAL CONTEXT

A. *RESTful Web Services*

Web Services can be defined as a way to let applications exchange information with Web servers [5]. In this interaction, the information exchanged may be contained inside documents written in a machine-readable format. The most used formats are Extensible Markup Language (XML) and JavaScript Object Notation (JSON) [6].

Currently, the main architecture of Web services is the Remote Procedure Call/Simple Object Access Protocol (RPC/SOAP), which is a World Wide Web Consortium (W3C) standard for the information exchange between Web services. The RPC/SOAP architecture is XML based, which ensures interoperability regardless of the technology used by the parts [7]. This is the central advantage of this architecture front of its predecessors, like Distributed Component Model (DCOM), Common Object Request Broker Architecture (CORBA) and Java Remove Method Invocation (RMI). Its predecessors were developed in specific technologies which difficult the communication between parts of services [5]. The RPC/SOAP, must be encapsulated within a standard format package named as SOAP envelope and must define a contract containing the communication rules. This contract is is defined in the Web Services Description Language (WSDL) document [8], which performs, among other functions, the syntactic description of the Web service elements. However, the RPC/SOAP architecture also has drawbacks. The excessive use of envelopes difficult the traffic and bring the addition of unnecessary computations, low performance and poor scalability [9].

In this context, the REST architecture is gaining importance, especially in the era we live in, the Web 2.0 [6][9]-[12]. This architecture was created by Roy Fields [2], one of the HTTP protocol creators. The main advantage of the REST architecture is the fact that communication occurs directly on the HTTP layer, without encapsulation need or use of envelopes, and it uses the basic elements of the protocol, like verbs and status codes. REST architecture focuses on resources, not on procedure calls or services, and it is an interesting approach for applications where the focus on interoperability is more important than the formal contract between parts.

Richardson and Ruby [13] have created the term RESTful to describe Web services that follow the REST paradigm and also created the Resource-Oriented Architecture (ROA) to define the architecture that faithfully follow the concepts and properties, defined by [2].

A resource is any real-world entity exposed on the internet and accessible by a Uniform Resource Identifier (URI). The URI is responsible for distinguish a resource from other. The resource can be a text, image, or even a device. The representation is the state of a resource at a given moment, in other words, representations are some data that represents a resource and are serialized to a machine-readable document like XML or JSON [12]. The representations are stateless, which means that each transaction does not keep information related to the previous transaction. This issue is solved using the concept of connectivity, which means that each representation has a link to the subsequent representation. The resources must provide a uniform interface for its handling, in other words, must always be accessed through the same URI, changing only the HTTP verbs. The most used verbs are POST, GET, PUT and DELETE, respectively associated to Create, Read, Update and Delete operations (also known as CRUD operations).

The syntactic description of RESTful Web service is optional and there still no consensus about what language should be standard [11][14]. The Web Application Description Language (WADL) is gaining notoriety despite the new 2.0 version of WSDL, which can also be used to syntactically describe this kind of service. According to Richardson et al. [13], WADL is “the most simple and elegant solution” to solve this problem.

B. *RESTful Implementation*

RESTful provides a multi-tier Web architecture composed by the Client, Application and Data tiers. The independence of these tiers provides flexibility to these dynamic Web applications. Divers programming languages, frameworks and styles can be used for developing each tier [13]. In addition, the Services can be developed in a specific language, such as Java, Python, PHP, Perl or Ruby and be consumed by an application written in a different language.

The Application tier provides a Presentation layer, a Business Logic layer and a Database Connector layer. The Business Logic intermediates between Presentation and Database Connector. Data is provided to the Presentation layer in the structure of objects. For example, in the Rails technology, services send and accept representations of active objects. These services map URIs to Rails controllers, Rails controllers to resources and resources to active objects. Despite the diversity of technologies available to develop RESTful Web Services, few of them are interoperable.

The Java platform has been the most powerful, flexible and user friendly platform for implementing the RESTful Web Services Application tier [15]. The development of RESTful Web services with Java is possible since 2008, when a new specification known as JAX-RS (The Java API for RESTful Web Services) [16] was defined to facilitate the implementation of such services. JAX-RS is based on metadata grammar of JDK 5, supporting the standardization of the RESTful services implementation. Today, JAX-RS is part of JavaEE 6 [17].

However, there exist several frameworks and APIs to implement RESTful services in Java as RESTEasy, Restlet, Struts2, Grails, Axis2, Certia4, sqlREST, REST-art [15].

The variety of programming languages also prevents reuse of the Business Logic implementation and reduces the interoperability due to the different formats and specifications used to implement the services.

C. *Model Driven Approach*

Model-Driven Engineering (MDE) is a software engineering approach that has gained significance in recent years. In MDE, the model is the central figure in the development of applications. The source-code is automatically generated since the application of transformation rules on pre-defined models [1].

By definition, the models are abstract entities that represent various aspects presented in the software, such as structure, behavior and graphical user interface [18]. MDE has two main approaches: MDA from the Object Management Group (OMG) and Eclipse Modeling Framework (EMF) from the Eclipse Foundation. The EMF provides a development framework which supports the MDA-based approach. We have been using concepts and resources of both approaches.

Figure 1 shows how the MDA approach involves the use of modeling languages, abstraction levels and independence of platform and programming languages. The M0 layer represents the real-world objects. The M1 layer is a model representation of the previous layer, represented by a modeling language. The models in M1 layer are defined using concepts described by metamodels in M2 layer. Each metamodel of M2 layer determines how expressive models can be. Analogously, metamodels are defined using concepts described by meta-metamodels in the M3 layer [19].

The Unified Modeling Language (UML) and Enterprise Distributed Object Computing (EDOC) are examples of modeling languages. At the M2 level UML and EDOC are defined by their metamodels which represent the elements of the structure of the modeling language. At the M3 level, we can use the OMG's Meta Object Facility (MOF) language or the Ecore language defined by the EMF, as depicted by the Figure 1.

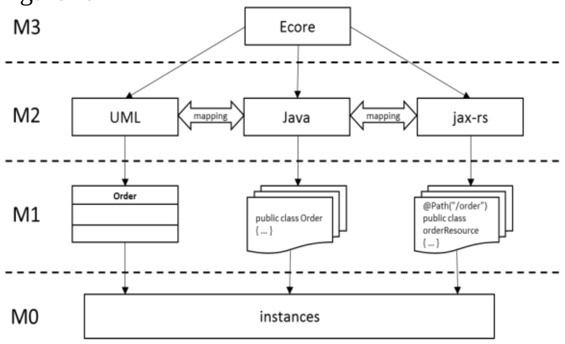


Figure 1. MDA's Abstraction Levels

In MDA, transformations are performed from source models to target models according to mappings, which are created by the identification of semantic correspondences between elements present in both models. The transformations rules are defined based on these mappings, executed by a transformation engine (e.g., the EMF engine) and written using transformation languages, such as Atlas Transformation Language (ATL) or OMG's Query View and Transform Language (QVT).

The ATL language (chosen in this study) provides a simple Object Constraint Language (OCL) based declarative language that facilitates the definition of transformation rules and it is available in a toolkit format to be used together with Eclipse [20].

The Eclipse Modeling Framework is an integration tool that uses class diagrams to represent metamodels and supports creation, storing, changing and opening model instances in XML Metadata Interchange (XMI) format. EMF unifies three important technologies: Java, XML and Unified Modeling Language (UML) with the Eclipse Integrated Development Environment (IDE) [19].

The main advantages of the MDA approach are portability, interoperability, reusability and technology independence, acting on the architectural concepts of separation between specification and implementation of software [21]. Thus, software engineers no longer need to worry about details of implementation language, focusing on the business rules and minimizing the occurrence of errors. The development of models containing only the business logic independent of technological details (platform, programming languages, and architectures) makes the software more portable. These business models can be mapped to many platforms only by the creation of new transformation rules [1].

III. MODEL DRIVEN RESTFUL WEB SERVICES

Fokaefs et al. [22] discuss the interoperability issue raised when two services using different architectures, like RPC/SOAP and REST, need to exchange information. The proposed solution was a metamodel, that abstracts architecture details and focuses only on the service elements.

We have discussed in [4] the problem of interoperability between syntactic and semantic description of RESTful Semantic Web services against the various existing languages. We have presented a model-driven approach, specifically on the creation of a metamodel and transformation rules in order to generate automatically the required documents that make the description of such services independently of the chosen language.

We have presented a metamodel named as RESTful Semantic Web Service (WSSR), which abstracts information present in the RESTful Web services and, to exemplify, we defined transformation rules that generated the syntactic description in WADL and the semantic description in Ontology Web Language for Services (OWL-S).

This work presented a solution of interoperability between syntactic and semantic description. However, it was not addressed the problem of interoperability also present in the Application tier of RESTful Web services, due to the fact of the wide variety of ways to implement these applications.

RPC/SOAP and RESTful Web services implemented in different frameworks and by different languages and formats are not interoperable and have many restrictions in their designs.

Some efforts have been made to design interoperable Web Services and model driven approaches have been applied as a solution to this problem. Some works can be found in the literature applying MDE-based techniques for developing RPC/SOAP Web Services. By the separation of concerns and by the development on different abstraction levels, Web Services metamodels can represent different tiers independent of technologies and programming languages then mapped to a target platform.

In this paper, we present an approach for the model driven development of the RESTful Web Service Application tier.

Figure 2 shows the architecture of our approach. The WSSR metamodel can be mapped to a semantic description language (A), to a syntactic description language (B) and to an implementation language (C). The (A) and (B) mappings were presented in [4] and in this paper, we discuss (C).

Through the solution proposed in this paper, it is possible to define the implementation logic of a RESTful Web Service independent of programming language, then by mapping rules and transformation process target different platforms without rewriting of the service implementation logic.

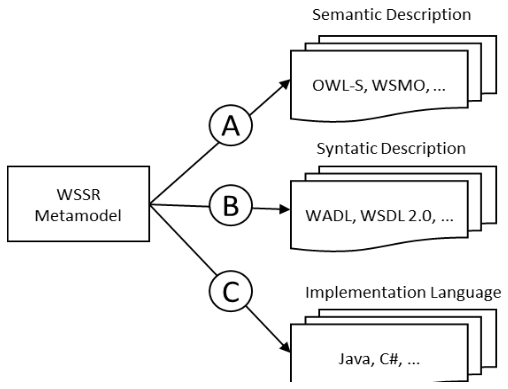


Figure 2. Possible transformations from WSSR metamodel

We provide the WSSR metamodel, which is responsible for defining the service business logic in the Application tier, abstracting platform (technologies, programming languages, etc) details. Then, a specific technology metamodel must be defined as the target platform chosen by the programmer to implement the service. Further, the identification of correspondences, named as mapping operation, will be defined between both metamodels. The mapping operation provides the correspondences needed to describe the transformation rules. The transformation rules, written in a transformation language, define which elements from a source metamodel will be transformed in which elements of the target metamodel. The transformation rules are applied in the model level, i.e., in instances of services. A source model must be conforms to the WSSR and the target model will be generated by the transformation engine, and it conforms to its target metamodel.

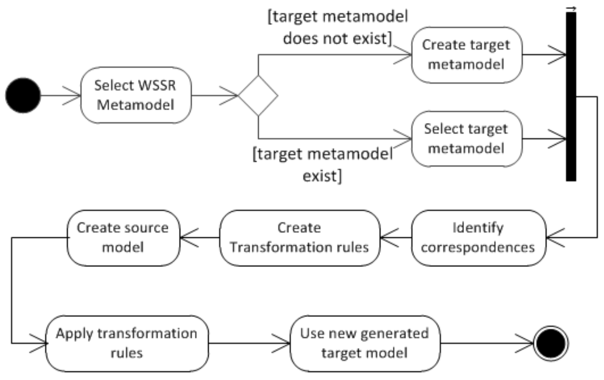


Figure 3. UML Activity Diagram of the Approach

Figure 4 illustrates, using a UML class diagram, the identification of semantic correspondences (mapping) of a fragment of our WSSR metamodel with the JAX-RS metamodel. The JAX-RS [16] specification was chosen because it is the standard specification for developing RESTful Web services in Java. This mapping allows the creation of the new transformation rules that will result in the implementation of the service in a fast and automated way.

In this figure, it is possible to see on the left side the WSSR metamodel with its main classes. The classes destined to do the semantic description were suppressed, because it is not the focus of this paper. It is important to note that the same WSSR metamodel – without any inclusion – can also be mapped to any language that implements RESTful Web services. The main class of the metamodel, named as *RESTService*, represents a RESTful service and contains the attributes necessary for the identification of such services, like URI, name, description, etc. The RESTful service has resources (*Resource class*), that are source of representations (*Representation class*). The resources are accessed by their methods (*Method class*) though HTTP methods (*HTTPMethod attribute*), which can have the values predefined in the *HTTPMethods* enumeration. The RESTful service is based on the paradigm of HTTP request and response, here represented by the *Request class and Response class*. The responses may be presented as representation format, and the requests are accessed through parameters (*Parameter class)*. These parameters may previously have established values, which are the options (*Option class*).

On the right side, can be seen the JAX-RS metamodel, which is an abstraction of the JAX-RS specification. This specification is present in the JavaEE and it is responsible for implementing RESTful Web services. The JAX-RS uses some Java language elements, such as packages, classes, methods and parameters, respectively represented by the *JPackage*, *JClass*, *JMethod* and *JParameter* classes. The *GET*, *POST*, *PUT* and *DELETE* classes are specializations of the *JMethod* class and inform which HTTP methods are related. The *JClass*, *JMethod* and *JParameter* classes are associated with the *JValue* class to combine Java annotations with themselves.

Between the two metamodels, each arrow identified by a circle represents the identification of correspondence between elements present in both WSSR and JAX-RS metamodel. The *R2C* arrow means that each *Resource class* corresponds to a *JClass class*. The *P2P* arrow means that each *Parameter class* corresponds to a *JParameter class*. Each *Method class* corresponds to a distinct class in the JAX-RS metamodel, depending on the *HTTPMethod* associated with it. If the *HTTPMethod* is GET, so the correspondence is between *Method class* and *GET class*, corresponding to the element M2Ge. Analogously, when the *HTTPMethod* is POST, PUT and DELETE, the *Method class* will be respectively associated with *POST*, *PUT* and *DELETE* classes, corresponding to elements M2Po, M2Pu and M2De.

This paper aims to generate an automated source-code generator related to the implementation of a RESTful Web service in any programming language that implements such services. To accomplish this, the WSSR metamodel must be mapped, by the correspondence identification, to the chosen programming language. Therefore, we chose the Java language with JAX-RS specification to exemplify the implementation.

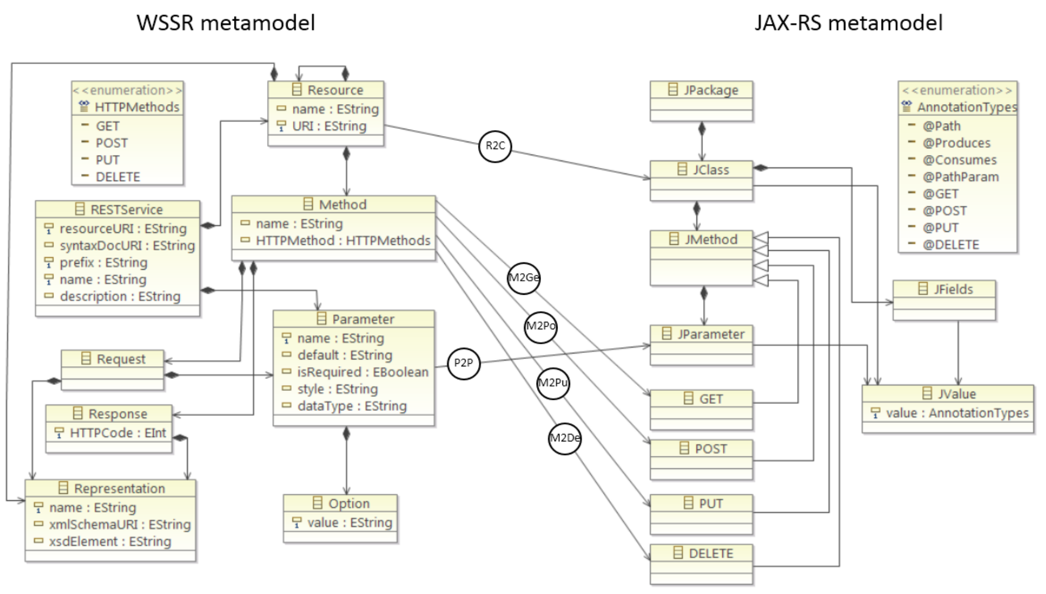


Figure 4. Correspondences between WSSR and JAX-RS metamodel elements

Figure 5 shows the transformation rules that were created using the elements that have the correspondences identified. The transformation rules where written using the ATL language and will transform the source model (conforms to the WSSR metamodel) to a target model (conforms to the JAX-RS metamodel). The target model will be the source code, i.e., the implementation of the RESTful Web service.

As can be seen, between lines 1 and 6, the code snippet refers to the creation of the resources. The code does an iteration over each method of the resource, calling a helper that corresponds to the creation of the methods between the lines 8 and 60. In this code snippet, also can be seen in lines 10, 30, 38 and 51 a condition that, according to the *HTTPMethod* associated with the method in the source model, will create a Java method with its corresponded annotation, such as the path and parameter that belong to this method.

```
01 helper context METAMODEL!Resource
02 def: toString(): String=
03  '@Path("/' + self.name + '")   ' +
04  'public class ' + self.name + 'Resource {   ' +
05  self.methods->iterate(i; acc: String='' | acc + i.Methods(self)) +
06 '}';
07
08 helper context METAMODEL!Method
09 def :Methods(resource: METAMODEL!Resource) : String =
10  if self.HTTPMethod.toString()='GET' then
11     '@GET' +
12     '@Produces("text/xml")' +
13     'public ArrayList<'+resource.name+'> get'+ resource.name +
14     'List('+self.methodRequests->iterate(i; acc : String = '' | acc +
15     i.requestQueryParameters->iterate(j; acc : String = '' | acc +
16     '@QueryParam("' + j.name + '") ' +
17     thisModule.convertDataType(j.dataType) +
18     resource.name + j.name + ', ')) + ') {  } ' +
19     '@Path("{' + thisModule.defaultParameter(self, resource) +
20     ' //return code }")' +
21     '@GET' +
22     '@Produces("text/xml")' +
23     'public ' + resource.name + ' get' + resource.name + '(' +
24     '@ParamPath("{' +
25     thisModule.defaultParameter(self, resource) + '}") ' +
26     thisModule.defaultDataType(self) +
27     thisModule.defaultParameter(self, resource) + ')
28  { //return code }'  else '' endif +
29
30  if self.HTTPMethod.toString()='POST' then
31     '@POST' + '   ' +
32     '@Consumes("text/xml")' +
33     '@Produces("text/xml")' +
34     'public ' + resource.name + ' post' + resource.name +
35     '(' + resource.name + ' ' + resource.name + ') ' +
36  '{ //return code }' else '' endif +
37
38  if self.HTTPMethod.toString()='PUT' then
39     '@Path("{'+thisModule.defaultParameter(self,resource)+'}")'+
40     '@PUT   ' +
41     '@Consumes("text/xml")' +
42     '@Produces("text/xml")' +
43     'public ' + resource.name + ' put' + resource.name + '(' +
44     '@ParamPath("{' +
45     thisModule.defaultParameter(self, resource) + '}") ' +
46     thisModule.defaultDataType(self) +
47     thisModule.defaultParameter(self, resource) + ', ' +
48     resource.name + ' ' + resource.name + ') ' +
49  '{ //return code }' else '' endif +
50
51  if self.HTTPMethod.toString()='DELETE' then
52     '@Path("{'+thisModule.defaultParameter(self,resource)+'}")'+
53     '@DELETE' + '   ' +
54     'public void del' + resource.name + '(' +
55     '@ParamPath("{' +
56     thisModule.defaultParameter(self, resource) + '}") ' +
57     thisModule.defaultDataType(self) + '
58     thisModule.defaultParameter(self, resource) + ', '
59     resource.name + ' ' + resource.name + ') '  +
60  '{ //return code }' else '' endif;
```

Figure 5. ATL snippet code of WSSR to JAX-RS transformation

By the specification of the correspondences between the source and target languages, the same code can be mapped generating different RESTful Web services implementations.

IV. CASE STUDY

As a case study, this paper proposes the implementation of a RESTful Web service that performs a simple product purchase order. The service must enable the order to be created, read, updated, and deleted, according to the concepts of the REST architecture described in the technological context of this paper.

To apply a model-driven approach, aiming to the implementation of the service, a model must be created conforms to the WSSR metamodel, previously presented. The model of the purchase order can be seen as an instance of the metamodel WSSR.

The model is depicted in Figure 6. Note that the model does not make use of all elements contained in the WSSR metamodel. Elements related to server Response (*Response class*) and the representation of the resource (*Representation class*), referred to the HTTP return code on the operations and the payload corresponding to representation sent and received from the server are not covered in this paper.

Thus, between lines 2 and 26 can be seen a RESTful Web service named *Order Service*, which have a resource named as *order*. Between lines 3 and 12 is the *GET method* and parameters that compose an application (code, product and quantity). The methods are responsible to perform operations on this resource. The *POST method* (line 13) refers to an operation of creating a new item. Between the Lines 14 and 19 can be found the *POST method*, which accepts the code parameter related to the order code that will be updated. Finally, between the lines 20 and 26 is depicted the *DELETE method*, which excludes the item passed by parameter.

```
01 <wssr:RESTService  name="Order Service"
02  <resources name="order" URI="order">
03   <methods name="Order-GET">
04      <methodRequests>
05         <requestQueryParameters name="code" default="true"
06         isRequired="true" style="query" dataType="xsd:int"/>
07         <requestQueryParameters name="product"
08         isRequired="true" style="query" dataType="xsd:string"/>
09         <requestQueryParameters name="quantity"
10         isRequired="true" style="query" dataType="xsd:int"/>
11    </methodRequests>
12   </methods>
13   <methods name="Order-POST" HTTPMethod="POST"/>
14   <methods name="Order-PUT"  HTTPMethod="PUT">
15     <methodRequests>
16      <requestQueryParameters name="code" default="true"
17      isRequired="true" style="query" dataType="xsd:int"/>
18    </methodRequests>
19   </methods>
20   <methods name="Order-DELETE" HTTPMethod="DELETE">
21     <methodRequests>
22        <requestQueryParameters name="code" default="true"
23        isRequired="true" style="query" dataType="xsd:int"/>
24     </methodRequests>
25   </methods>
26  </resources>
```

Figure 6. A sample model conforms WSSR metamodel

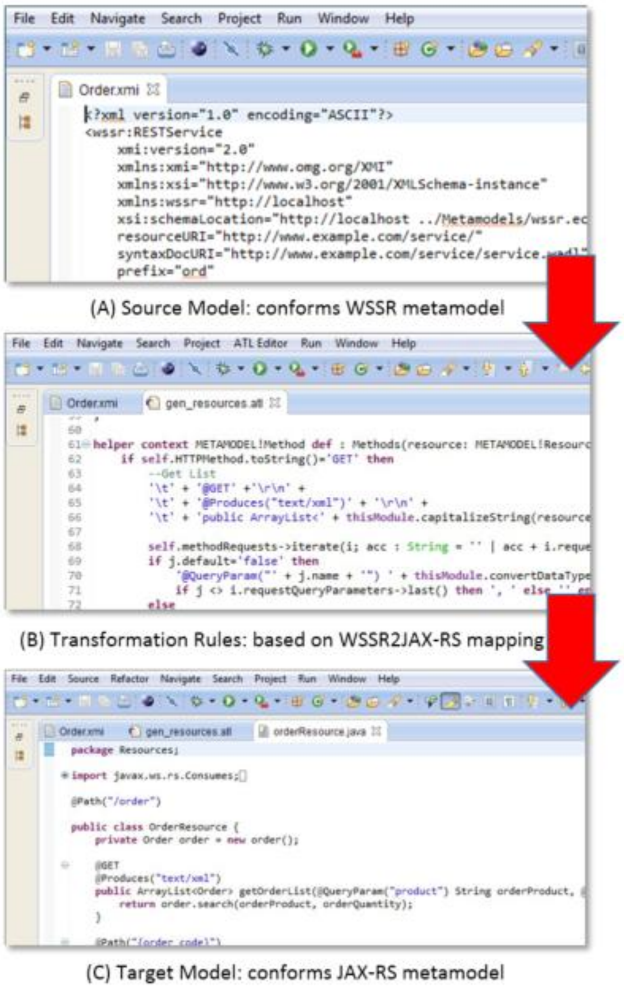
The execution of the transformation rules, created in the previous chapter, results in a source code written in Java and using the JAX-RS specification, as shown in Figure 7.

The figure shows the *orderResource* class containing methods and parameters appropriately associated with the Java annotations that will transform these elements in resources, methods and parameters of a RESTful Web service.

```
@Path("/order")
public class orderResource {
 @GET
 @Produces("text/xml")
 public ArrayList<order> getOrderList(
    @QueryParam("code") int orderCode,
    @QueryParam("product") String orderProduct,
    @QueryParam("quantity") int orderQuantity)
 { //return code }

 @Path("{order_code}")
 @GET
 @Produces("text/xml")
 public order getorder(
    @ParamPath("{order_code}") int orderCode)
 { //return code }

 @POST
 @Consumes("text/xml")
 @Produces("text/xml")
 public order postOrder(Order order)
 { //return code }

 @Path("{order_code}")
 @PUT
 @Consumes("text/xml")
 @Produces("text/xml")
 public order putOrder(
 @ParamPath("{order_code}") int orderCode, Order order)
 { //return code }

 @Path("{order_code}")
 @DELETE
 public void delOrder(
 @ParamPath("{order_code}") int orderCode, Order order)
 { //return code }
}
```

Figure 7. ATL snippet code of WSSR to JAX-RS transformation

Figure 8 shows a use case example developed through our approach, which is implemented in the Eclipse EMF. EMF has been the most used framework for the development of model driven approaches. It provides plugins for defining models, transformation language APIs, transformation engines and different modeling languages. In part (A) of the figure, we present the implementation of the source model, which conforms to the WSSR metamodel, previously defined in the EMF. The transformation rules are defined in the ATL language, as shown in part (B). By the interpretation of the transformation rules, the ATL transformation engine will generate, in a semi-automatic way, the target code of the RESTful Web Service in the Java language, as shown in part (C). The parts (A), (B) and (C) represent the implementation of the source code (fragment) depicted in the Figures 5, 6 and 7. The same source model can generate different services in different languages providing the interoperability proposed.


File Edit Navigate Search Project Run Window Help
Order.xmi
(A) Source Model: conforms WSSR metamodel
File Edit Navigate Search Project ATL Editor Run Window Help
Order.xmi gen_resources.atl
(B) Transformation Rules: based on WSSR2JAX-RS mapping
File Edit Source Refactor Navigate Search Project Run Window Help
Order.xmi gen_resources.atl orderResource.java
(C) Target Model: conforms JAX-RS metamodel

```
<?xml version="1.0" encoding="ASCII"?>
<wssr:RESTService
    xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:wssr="http://localhost"
    xsi:schemaLocation="http://localhost ../Metamodels/wssr.ec
    resourceURI="http://www.example.com/service/"
    syntaxDocURI="http://www.example.com/service/service.wadl"
    prefix="ord"
```

```
helper context METAMODEL!Method def : Methods(resource: METAMODEL!Resourc
    if self.HTTPMethod.toString()='GET' then
        --Get List
        '\t' + '@GET' +'\r\n' +
        '\t' + '@Produces("text/xml")' + '\r\n' +
        '\t' + 'public ArrayList<' + thisModule.capitalizeString(resource

        self.methodRequests->iterate(i; acc : String = '' | acc + i.reque
        if j.default='false' then
            '@QueryParam("' + j.name + '") ' + thisModule.convertDataType
            if j <> i.requestQueryParameters->last() then ', ' else '' en
        else
```

```
package Resources;

import javax.ws.rs.Consumes;

@Path("/order")

public class OrderResource {
    private Order order = new order();

    @GET
    @Produces("text/xml")
    public ArrayList<Order> getOrderList(@QueryParam("product") String orderProduct,
        return order.search(orderProduct, orderQuantity);
    }

    @Path("{order_code}")
```

Figure 8. Development of the approach in EMF/Eclipse environment

V. CONCLUSION AND FUTURE WORK

The lack of interoperability between RESTful Web services have been discussed at the architectural level, syntactic description and semantic description. In this paper, we have discussed this issue at the implementation level. We have proposed a model-driven approach to solve the lack of interoperability of the Application tier of the REST architecture. We have presented a metamodel, mapping specifications and the transformation rules targeting the Java language, but the same approach can target any language that implements RESTful Web services. Comparing to the prior research, this approach provides some benefits beyond interoperability such as agile development, standardization, reuse and focus on the business rules. As future work, the WSSR metamodel may be mapped to others languages or specifications that are also used to implement RESTful Web services and generate the correspondent source code.

REFERENCES

[1] I. Sommerville, “Software Engineering,” Pearson Prentice Hall, 2011.

[2] R. Fielding, “Architectural styles and the design of network-based software architectures,” University of California, 2000.

[3] D. Chen and G. Doumeingts, "European initiatives to develop interoperability of enterprise applications—basic concepts, framework and roadmap," Annual Reviews in Control, v. 27, n. 2, pp. 153-162, 2003.

[4] N. Tavares and S. Vale, “A Model Driven Approach for the Development of Semantic RESTful Web Services,” in 15th International Conference on Information Integration and Web-based Applications & Services (iiWAS2013), pp. 290-299, 2013.

[5] E. Cerami, “Web Services Essentials,” O’Reilly Media, 2002.

[6] O. Xavier, “Semantic Web Services based on RESTful”, Federal University of Goiás, 2001.

[7] M. Hadley, N. Mendelsohn, J. Moreau, H. Nielsen, and M. Gudgin, “SOAP Version 1.2 Part 1: Messaging Framework,” W3C REC REC-soap12-part1-20030624, June, 240-8491, 2003.

[8] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, “Web services description language (WSDL) 1.1,” W3C Note, 2001.

[9] O. Filho, “Semantic Services: A RESTful approach,” University of São Paulo, 2009.

[10] F. Valverde and O. Pastor, "Dealing with REST Services in Model-driven Web Engineering Methods," in V Jornadas Científico-Técnicas en Servicios Web y SOA, JSWEB, pp. 243-250, 2009.

[11] Y. J. Lee and C. S. Kim, “Building Semantic Ontologies for RESTful Web Services,” in Computer Information Systems and Industrial Management Applications (CISIM), International Conference, pp. 383-386, 2010.

[12] R. Khorasgani, E. Stroulia, and O. Zaïane, “Web service matching for RESTful web services,” in Web Systems Evolution (WSE), 13th IEEE International Symposium, pp. 115-124, 2011.

[13] L. Richardson and S. Ruby, “RESTful Serviços Web,” O’Reilly Media, 2007.

[14] C. Pautasso, “RESTful Web service composition with BPEL for REST,” in Data & Knowledge Engineering, v. 68, n. 9, pp. 851-866, 2009.

[15] H. Li, "RESTful Web service frameworks in Java," in Signal Processing, Communications and Computing (ICSPCC). IEEE International Conference, pp. 1-4, 2011.

[16] M. Hadley and P. Sandoz, "JAX-RS: Java™ API for RESTful Web Services," Java Specification Request (JSR) 311, 2008, <https://jcp.org/en/jsr/detail?id=311> 23.10.2013.

[17] Java, E. E. “At a Glance,” 2006, <http://www.oracle.com/technetwork/java/javaee> 23.10.2013.

[18] X. Qafmolla and V. C. Nguyen, “Automation of Web services development using model driven techniques,” in Computer and Automation Engineering (ICCAE), The 2nd International Conference on. IEEE International Conference, vol. 3, pp. 190-194, 2010.

[19] S. Staab, T. Walter, G. Gröner, and F. S. Parreiras, “Model driven engineering with ontology technologies,” in Reasoning Web, Semantic Technologies for Software Engineering. Springer Berlin Heidelberg, pp. 62-98, 2010.

[20] A. Radjenovic and R. Paige, "Behavioural interoperability to support model-driven systems integration," in First International Workshop on Model-Driven Interoperability. ACM, pp. 98-107, 2010.

[21] D. Amar Bensaber and M. Malki, “Development of semantic Web services: model driven approach,” in 8th international conference on New technologies in distributed systems. ACM, pp. 40, 2008.

[22] M. Fokaefs and E. Stroulia, "WSMeta: a meta-model for web services to compare service interfaces," in Proceedings of the 17th Panhellenic Conference on Informatics. ACM, pp. 1-8, 2013.