Advanced REST API Management and Evolution Using MDA

Marek Polák
Department of Software Engineering
Charles University in Prague
Malostranské nám. 25, Prague, Czech Republic
polak@ksi.mff.cuni.cz

Irena Holubová
Department of Software Engineering
Charles University in Prague
Malostranské nám. 25, Prague, Czech Republic
holubova@ksi.mff.cuni.cz

ABSTRACT

The REST (Representational State Transfer) has become a popular and preferred way of communication on the Web. JSON (JavaScript Object Notation) [5] is the most used data interchange format these days. JSON can be easily used with (but not-only) applications written in JavaScript or other programming languages. In our previous work we presented a solution how to describe REST API based on the MDA principle and how to provide automatic evolution management. The aim of this paper is to extend our previous results with deeper analysis and additional model operations to make evolution process simple for the API designer. Next, we aim to enlarge abilities of the platform for the specific model of the REST to be able to generate final communication structures in JSON, or other formats like, e.g., XML, as well as their schemas. The proposed approach was implemented within a complex evolution-management framework called *DaemonX* which provides an extensible platform for evolution management of various data formats, such as, e.g., XML, relational, object, etc. Several experiments proving the concept were then carried out.

Keywords

REST, JSON, MDA, Evolution Management

1. INTRODUCTION

The REST (Representational State Transfer) [10] has become a popular and preferred way of communication on the Web. And JSON (JavaScript Object Notation) [5] is the most used data interchange format these days. However, although these technologies are used widely over the Internet, there are no official standards for REST (like, e.g., the WSDL [8] for Web Services) and for JSON (like, e.g., the XML Schema [16] for XML [17] documents). There only exist several projects partially covering these topics for specific purposes, such as, e.g., RAML [6], Swagger [7], or HAL [4], which enable generating human/machine description (documentation) of the API, its routing, JSON content structure, etc. But a more important and difficult related issue is versioning of the APIs. The mentioned solutions, surprisingly, do not solve this problem at all. They do not provide any relations between two subsequent versions except for API designer/developer comments. This brings the need to manually check every new version by the API consumer and subsequent manual updates of the consumer's code to ensure compatibility. With the growing size of an application this task becomes highly difficult and error-prone.

In our previous paper [15] we presented an initial solution how to describe REST API based on the MDA (Model-Driven Architecture) principe [11] and how to provide automatic evolution management between subsequent API versions derived from the original version. *Evolution management* is a mechanism how to analyze and (semi)automatically propagate changes between related parts of the system using defined rules. We defined a model for REST resource representation called *Resource Model*, operations over this model and algorithms for propagation of changes into this model.

The aim of this paper is to extend the previous solution with deeper analysis, new model properties, additional model operations and extended algorithms to make the evolution process more advanced and simple for the API designer. Next, we extend the abilities of the platform for the specific model of the REST to be able to generate final communication structures in JSON, or other formats like, e.g., XML and their schemas too. The proposed approach was implemented within a complex evolution-management framework called *DaemonX* [9] which provides an extensible platform for evolution management of various data formats, such as, e.g., XML, relational, object, etc. Several experiments proving the concept were then carried out.

The paper is structured as follows: In Section 2 we analyze the related works. The MDA principle is briefly described in Section 3. In Section 4 we introduce our extended Resource Model. In Section 5 we describe the mapping and evolution process between the MDA PIM and the Resource Model. In Section 6 we provide the proof of the concept and we conclude the paper in Section 7.

2. RELATED WORK

As we have mentioned, except for our previous paper there are no existing papers solving a similar problem. Articles dealing with API versioning [3, 2] propose creation of a new API version and suggest a list of best practises such as how to prevent problems during the future changes and versions.

On the other hand, there are various solutions for REST

DChanges '2015 Lausanne, Switzerland
© 2015 ACM. ISBN 978-1-4503-3714-4/15/09...$15.00
DOI: http://dx.doi.org/10.1145/2881631.2881635

services management (documentation) like the API Blueprint [1], RAML [6], or Swagger [7]. All the mentioned frameworks describe the REST API so consumers can use the API in a comprehensible way. They also offer own description language, editors for API creation, documentation, stub (code) generation, etc. But none of them provides a way how to manage changes between API versions.

Efficient change management becomes a problem in larger projects as well as later during the maintenance phases of the software. In this situation, every change must be done precisely, correctly and completely to prevent errors. (Semi)automatic mechanism which helps to identify the affected parts for the developer or even performs the change automatically is then very important.

3. MODEL-DRIVEN ARCHITECTURE

The *Model-Driven Architecture* (MDA) [11] is an approach to system development. MDA deals with the idea of separating the specification of the operation of a system from details of the way how the system uses the capabilities of its platform. Basically, it consists of various layers. However, in this paper we will describe and use only a subset of these layers.

Platform-Independent Model.

A *platform-independent model* (PIM) is a view of the system from the platform-independent viewpoint. It focuses on the general operation of the system while hiding the details necessary for a particularly selected platform. PIM represents a part of the complete specification that does not change from one platform to another. A general-purpose modeling language like UML [14] is often used for modeling of PIMs. A basic platform-independent model consists of the following parts:

- ***Class*** – A class represents the main item of the model. Every class has a nonempty name, a collection of attributes, and a collection of functions.
- ***Attribute*** – Each attribute has a nonempty name and a data type. It belongs to exactly one class.
- ***Function*** – Every function has a name, a return type, and a collection of parameters. Every parameter has a name and a data type.
- ***Association*** – An association represents a connection of two classes. An association has a name and cardinalities used with the particular ends of the association.

Platform-Specific Model.

A *platform-specific model* (PSM) is a view of the system from the platform-specific viewpoint. It combines the platform-independent viewpoint with an additional focus on the details of the specific platform by a system, e.g., a database model, an XML model, an object model, etc.

In particular in this paper we will extend the platform-specific model of REST, as we will see in Section 4.

Model Transformation.

In general, a *model transformation* is a conversion of one model to another model of the same system. From the perspective of MDA, the most interesting one is the model transformation from PIM to PSM. This transformation can be in general described as follows:

1. The PIM is extended with special marks, which define general mapping rules.
2. A specific platform is chosen.
3. Transformations corresponding to the mapping marks are executed according to the chosen specific platform.

Mapping marks can be represented as (directed) lines between PIM items and PSM items.

4. RESOURCE MODEL

For the purpose of this paper we will use the platform specific model for the REST, called *Resource Model* (ReM), defined in [15]. Thanks to this we can model and visualize the resource structure in a user-friendly way. Over the model there are defined algorithms providing generation of the base code structure for particular platforms and the schemas for communication structures, e.g., the JSON schema. For the purpose of this paper, the model will be extended with attributes which define a value that can be attached to an instance.

Definition 1. *A* resource graph *G of a resource R is a directed graph $G_R = (V, E)$, where V is a set of* resource vertices *and E is a set of* resource edges. *For the resource graph it must hold that it is a tree.*

Every resource vertex $v \in V$ has a set of resource attributes P_v and a set of resource functions F_v. Each resource attribute *has a name, a data type, and a flag denoting if it is used as a key. Let D be a set of parameters, each having a name and a data type. Let D' be a set of resulting values, each having a data type. A* resource function *$f : 2^{(D)} \rightarrow D'$ provides a particular result $d \in D'$ for a particular subset of parameters from D.*

An example (and a DaemonX screen shot) of the ReM representing a simple e-shop diagram is depicted in Figure 1. Every word represents a vertex in the ReM or a vertex function. A word in the curly brackets represents a parameter. This parameter is applied as a function parameter or as a selector for an item from the collection. Specific usage and implementation depends on the author. A slash denotes a directed edge – from the left to the right. An example of respective generated resources can be as follows: `shop/user`, `shop/user/{:uid}`, `shop/user/{:uid}/order`, `shop/user/{:uid}/order/{:oid}`, `shop/user/{:uid}/order/{:oid}/item/{:iid}/setcount/{count}`.

Note that the same resources (e.g., `shop/user`) can be combined with different HTML methods (e.g., *GET*, *POST*, or *PUT*) for different purposes.

The list of the REST resources generated from this simple example has 21 items and thus any future update in the model (e.g., renaming of class *User* to *Customer*) can trigger many necessary changes which is a difficult and error-prone task. For this purpose, an ability to automatically propagate changes from the PIM to the corresponding ReM is essential. In the following section we describe how this problem can be solved fully and efficiently.

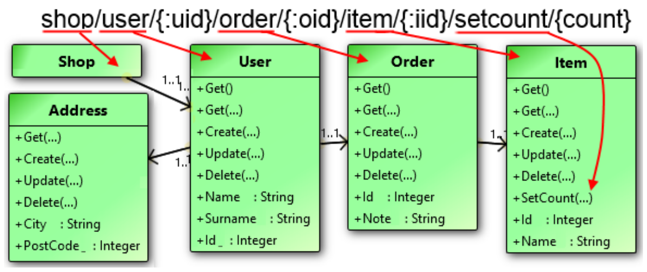

Figure 1: An example of the ReM of a simple e-shop. The green rectangles represent vertices, the black arrows represent edges of the ReM. The red arrows represent the mapping to a sample REST resource.

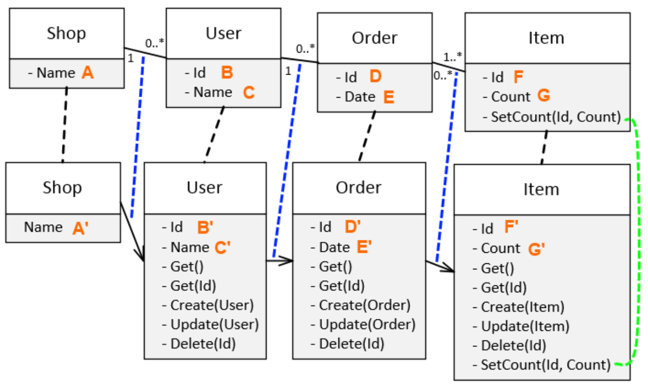

Figure 2: An example of the PIM-to-ReM mapping

5. EVOLUTION

To ensure evolution management, i.e. correct propagation of changes, we first need to define a mapping between the respective models. The mapping, or so-called *relation*, between the PIM and the ReM is relatively straightforward. The mapping is described as follows:

- **PIM Class → Resource Graph Vertex** A PIM Class is mapped directly to an ReM Vertex.
- **PIM Class Function → Resource Graph Vertex Function** A PIM Class Function is mapped to an ReM Vertex Function. Resource Graph Vertex Function represents an operation over the PIM Class instance.
- **PIM Association → Resource Graph Edge** A PIM Association is mapped to an ReM Edge. As in PIM Association in PIM, Resource Graph Edge represents a relation between Resource Graph Vertices.
- **PIM Attribute → Resource Vertex Attribute** A PIM Attribute is mapped to an ReM Attribute.

An example of a mapping between a PIM and an ReM is depicted in Figure 2. All classes of the PIM are mapped to the corresponding ReM vertices (the mapping is represented with the black dashed line). The only PIM class function is mapped to the corresponding ReM vertex function (marked with the green dashed line). PIM associations are mapped to edges in the ReM (marked with the blue dashed lines). Finally, PIM Class attributes are mapped to the corresponding attributes in ReM Vertices (marked with the corresponding orange letters).

First let us define atomic operations for both PIM and ReM (see Section 5.1 and 5.2). We will define only operations needed for the purpose of this paper. The full list of the operations defined over both models can be found in [15].

5.1 Atomic PIM Operations

Let $M_S = (C_S, I_S)$ be a PIM. C_S is a set of classes, I_S is a set of connections, where $I_k(C_l, C_m)$, $k \in [1, n]$ is a connection between classes C_l and C_m and has name I_{k_N}. Each class $C_i \in C_S$, $i \in [1, v]$ has a name C_{i_N}, a set of attributes C_{i_P} and a set of functions C_{i_F}. Every attribute P_j, $j \in [0, s]$ has a name P_{j_N} and a type P_{j_T}. Every function F_o, $o \in [0, t]$ has a name F_{o_N}, a return type F_{o_T}, and a set of parameters F_{o_R}. Each function parameter R_q, $q \in [0, u]$ has a name R_{q_N} and a type R_{q_T}.

PIM atomic operations of attributes are defined as follows:

- **Attribute Creating** ($\gamma_P : (C_i, P_j) \to C_i'$): The operation adds attribute P_j to class C_i. It returns class C_i', where $C_{i_N}' = C_{i_N}$, $C_{i_F}' = C_{i_F}$, and $C_{i_P}' = C_{i_P} \cup \{P_j\}$, e.g., only the set of attributes is changed. ***Precondition:*** Attribute P_j must have name P_{j_N} and type P_{j_T}. Class C_i must exist. There must not exists attribute P_l, $P_l \in C_i$, $P_l \neq P_j$ where $P_{l_N} = P_{j_N}$.
- **Attribute Renaming** ($\alpha_P : (P_j, m) \to P_j'$): The operation returns attribute P_j', where $P_{j_N}' = m$, $P_{j_T}' = P_{j_T}$. ***Precondition:*** Attribute P_j must exist. There must not exists attribute P_l, $P_l \in C_i$, $P_j \in C_i$, $P_l \neq P_j$ where $P_{l_N} = m$.
- **Attribute Type Changing** ($\beta_P : (P_j, t) \to P_j'$): The operation returns attribute P_j', where $P_{j_T}' = t$, $P_{j_N}' = P_{j_N}$. ***Precondition:*** Attribute P_j must exist.
- **Attribute Removing** ($\delta_P : (C_i, P_j) \to C_i'$): The operation removes attribute $P_j \in C_{i_P}$ from class C_i. It returns class C_i', where $C_{i_N}' = C_{i_N}$, $C_{i_P}' = C_{i_P}$, $C_{i_F}' = C_{i_F}$, and $C_{i_P}' = C_{i_P} \setminus \{P_j\}$. ***Precondition:*** Class C_i and attribute P_j must exist.
- **Attribute Moving** ($\epsilon_P : (C_i, C_j, P_k) \to (C_i', C_j')$): The operation moves attribute $P_k \in C_{i_P}$ from class C_i to class C_j. It returns class C_i', where $C_{i_N}' = C_{i_N}$, $C_{i_F}' = C_{i_F}$, and $C_{i_P}' = C_{i_P} \setminus \{P_k\}$ and class C_j', where $C_{j_N}' = C_{j_N}$, $C_{j_F}' = C_{j_F}$, and $C_{j_P}' = C_{j_P} \cup \{P_k\}$. ***Precondition:*** Classes C_i, C_j and attribute P_k must exist. There must not exist attribute P_l, $P_l \in C_j$, $P_l \neq P_k$ where $P_{l_N} = P_{k_N}$.

The specified pre-conditions ensure, that the operations transform the schema from one consistent status to another. The consistency represents the correspondence to the original definition of the schema.

5.2 Atomic ReM Operations

Let $M_R = (V_R, E_R)$ be an ReM. Each vertex $V_i \in V_R, i \in [1, n]$ has a name V_{i_N}, a set of functions V_{i_F}, and a set of attributes V_{i_P}. Every function F_j, $j \in [0, o]$ has a name F_{j_N}, a return type F_{j_T}, and a set of parameters F_{j_R}. Each function parameter R_k, $k \in [0, m]$ has a name R_{k_N} and a type R_{k_T}. Each attribute P_l, $l \in [0,t]$ has a name P_{l_N}, a type P_{l_T}, and a boolean flag P_{l_B} which denotes if it is used as an identifier or not. E_R is a set of edges, where $E_l(V_p, V_q)$, $l \in [1,s]$, $E_l \in E_R$ is an ordered edge between vertices V_p and V_q.

The extended set of attributes operations is defined as follows:

- **Attribute Creating** ($\gamma_P : (V_i, P_l) \rightarrow V_i'$): The operation adds parameter P_l to vertex V_i. It returns vertex V_i', where $V_{i_N}' = V_{i_N}$ and $V_{i_P}' = V_{i_P} \cup \{P_l\}$. ***Precondition:*** Vertex V_i must exist. Attribute name P_{l_N} and type P_{l_T} must be set. There must not exists attribute P_j, $P_j \in V_i$, $P_j \neq P_l$ where $P_{l_N} = P_{j_N}$.
- **Attribute Removing** ($\delta_P : (V_i, P_l) \rightarrow V_i'$): The operation removes attribute $P_l \in V_{i_P}$ from vertex V_i. It returns vertex V_i', where $V_{i_N}' = V_{i_N}$ and $V_{i_P}' = V_{i_P} \backslash \{P_l\}$. ***Precondition:*** Vertex V_i and attribute P_l must exist.
- **Attribute Renaming** ($\alpha_P : (P_l, m) \rightarrow P_l'$): The operation returns attribute P_t', where $P_{l_N}' = m$, $P_{l_T}' = P_{i_T}'$, and $P_{l_B}' = P_{l_B}$. ***Precondition:*** Attribute P_l must exist. ***Postcondition:*** There must not exists attribute P_j, $P_j \in C_i$, $P_l \in V_i$, $P_l \neq P_j$ where $P_{j_N} = m$.
- **Attribute Type Changing** ($\beta_P : (P_l, u) \rightarrow P_t'$): The operation returns attribute P_l', where $P_{l_T}' = u$, $P_{l_N}' = P_{l_N}$, and $P_{l_B}' = P_{l_B}$. ***Precondition:*** Attribute P_t must exist.
- **Attribute Moving** ($\epsilon_P : (V_i, V_j, P_k) \rightarrow (V_i', V_j')$): The operation moves attribute $P_k \in V_{i_P}$ from vertex V_i to vertex V_j. It returns class V_i', where $V_{i_N}' = V_{i_N}$, $v_{i_F}' = V_{i_F}$, and $V_{i_P}' = v_{i_P} \backslash \{P_k\}$ and vertex V_j', where $V_{j_N}' = V_{j_N}$, $V_{j_F}' = V_{j_F}$, and $V_{j_P}' = V_{j_P} \cup \{P_k\}$. ***Precondition:*** Vertices V_i, V_j and attribute P_k must exist. There must not exists attribute P_l, $P_l \in V_j$, $P_l \neq P_k$ where $P_{l_N} = P_{k_N}$.

5.3 View Model

In real-world applications it is common that models defined in PIM do not fully correspond to the models defined in PSM. For instance, there can be different attributes in two corresponding classes, there may be a relation between classes in PIM not represented in PSM, or some classes from PIM might not be represented in the PSM. Due to this feature, it is common that in PIM a special class is created and this class projects only selected attributes or even attributes having different types than in the original class. These classes are usually called *View Model Classes*. For instance, suppose there exists PIM class *User* with attributes *Name*, *Id*, *Email*, *Created* and its corresponding PSM view model *User* only with attributes *Name* and *Email* – PIM attributes *Id* and *Created* are hidden from user view.

Thanks to the above defined extension of the ReM with attributes, it is possible to handle this requirement – the ReM in fact represents a view of the PIM.

5.4 Model Nesting

Another situation can be a combination of several PIM classes into one ReM class – so-called *nesting*. An example of the nesting is depicted in Figure 3, where PIM classes *Product* and *ProductVersion* are nested into one ReM class *Product*. This situation is very common, because the internal architecture and the structures of the system do not have to correspond to the structures offered by the API to the consumers.

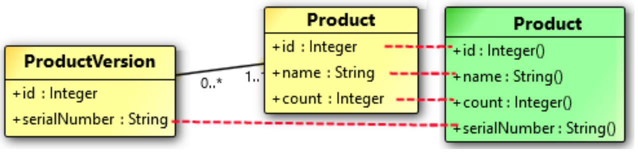

Figure 3: An example of nesting PIM classes *Product* and *ProductVersion* into ReM class *Product*

To be able to provide changes in the PIM classes and their related PSM classes, we need to define algorithms which solve potential problems, such as, e.g., attribute name collision. Figure 4 depicts a situation when attribute *year* is added to PIM class *Product* and this change is correctly propagated to ReM class *Product*. Figure 5 illustrates a situation of invalid propagation. Attribute *name* is added to PIM class *Product* and cannot be propagated to ReM class *Product*, because there already exists attribute *name*.

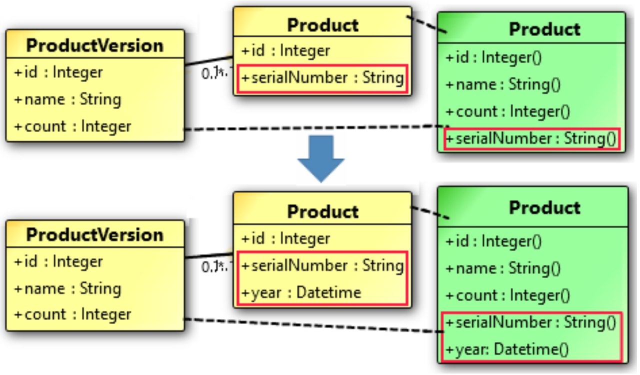

Figure 4: An example of correct propagation of adding attribute *year* to class *Product*

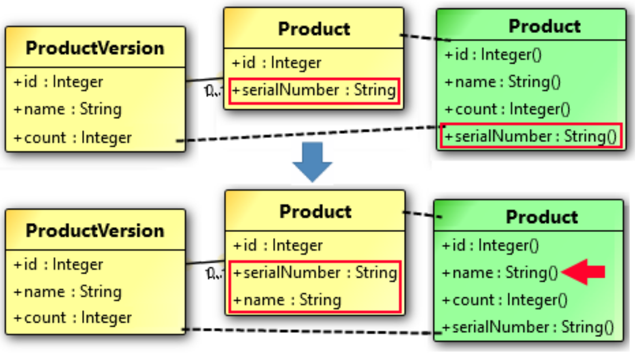

Figure 5: An example of invalid propagation of adding attribute *name* to class *Product*

5.5 Resource Parameters Evolution

In the REST resource, particular data can be specified by its unique identifier (e.g., ID or hash). This identifier is specified after the concrete data item. For example, in the resource `shop/user/{:uid}/order/{:oid}`, identifier `{:uid}` specifies ID of the user and `{:oid}` specifies id of the order. It is possible to specify more attributes as identifiers (or keys). In most cases names of these identifiers are mapped directly to the class attributes. So a change of the attribute name influences the resource and correct change propagation must be provided as well.

5.6 Propagation Algorithms

In this section new propagation algorithms from PIM to ReM are described. Only one atomic operation is every time called over PIM, but the propagation algorithm can trigger multiple atomic operations – i.e. a so-called *composite* operation – over the ReM to ensure proper model state.

Attribute Creating.

Algorithm 1 adds new attribute P to PIM class C. Next, it tries to add the newly created attributes to all related vertices of class C in the ReM. There must be done a check if there already exists an attribute with the same name by method $ExistsAnotherAttributeWithSetName(Attribute, TargetClass)$. If so, an exception is raised and the algorithm ends. Finally, the algorithm creates relations between attribute P and attributes created in the ReM using function $CreateRelation(PIMModelItem, ResourceModelItem)$.

Algorithm 1 AttributeCreating

```
Require: Attribute $P$ to be added to class $C$
Ensure: Creating of the attributes in related vertices of the class
 1: $CreateAttributeClass(C, P)$
 2: $relatedVertices \leftarrow GetRelationsOf(C, 'VertexType')$
 3: for all $rv \in relatedVertices$ do
 4:     if $ExistsAnotherAttributeWithSetName(rv, P.Name)$ then
 5:         // raise a warning
 6:         $raise\ AttributeWithNameAlreadyExists(N)$
 7:     else
 8:         $p \leftarrow newAttribute(P.Name)$
 9:         $CreateAttribute(rv, p)$
10:         $CreateRelation(P, p)$
11:     end if
12: end for
```

Attribute Removing.

Algorithm 2 removes a given attribute P from its class. Additionally it removes all related attributes in the ReM and relations by function $RemoveAttribute(ResourceModelAttribute)$.

Algorithm 2 AttributeRemoving

```
Require: Attribute $P$ to be removed
Ensure: Removing of the attribute $P$ and related attributes in ReM
 1: $relatedAttributes \leftarrow GetRelationsOf(P, 'AttributeType')$
 2: $RemoveAttribute(P)$
 3: for all $rp \in relatedAttributes$ do
 4:     $RemoveAttribute(rp)$
 5: end for
```

Attribute Type Updating.

As described in Algorithm 3, the algorithm first changes type T of attribute P. Next, it changes the type of all related attributes in the ReM.

Algorithm 3 AttributeTypeUpdating

```
Require: Attribute $P$, new type $T$ of the function
Ensure: Update of the attribute type and its related attributes in ReM
 1: $P.Type \leftarrow T$
 2: $relatedAttributes \leftarrow GetRelationsOf(P, 'AttributeType')$
 3: for all $rp \in relatedAttributes$ do
 4:     $rp.Type \leftarrow T$
 5: end for
```

Attribute Renaming.

Algorithm 4 gets as parameters the attribute and a new name of the attribute. First, it changes the name of the PIM attribute. Next, it finds all related attributes of attribute P. For every attribute in the ReM, there must be checked that there is not an attribute with the same name (by method $ExistsAnotherAttributeWithSetName()$). If so, an error is raised.

Algorithm 4 AttributeRenaming

```
Require: Attribute $P$, new name $N$ of the attribute
Ensure: Renaming of the related attribute in vertex
 1: $P.Name \leftarrow N$
 2: $relatedAttributes \leftarrow GetRelationsOf(P, 'AttributeType')$
 3: for all $rp \in relatedAttributes$ do
 4:     if $ExistsAnotherAttributeWithSetName(rp, N)$ then
 5:         // raise a warning
 6:         $raise\ AttributeWithNameAlreadyExists(N)$
 7:     else
 8:         $rp.Name \leftarrow N$
 9:     end if
10: end for
```

Attribute Moving.

Algorithm 5 gets as parameters the attribute to be moved, the class where the attribute is and the class to which the attribute should be moved. First, the algorithm checks if there exists an attribute with the same name in the target class using method $ExistsAnotherAttributeWithSetName(Attribute, TargetClass)$. If so, the algorithm raises an exception $AttributeWithNameAlreadyExistsInClass(Attribute, TargetClass)$ and ends. If not, the attribute is moved. Next, the algorithm tries to move all related ReM attributes to the ReM classes related to target PIM class. If there is no collision with the attribute name, the ReM attribute is moved. Otherwise the algorithm raises an error.

An example of the movement propagation is depicted in Figure 6. Attribute *assemblyYear* is moved from PIM class *Product* to class *ProductVersion* and subsequently moved in ReM.

Algorithm 5 AttributeMoving

```
Require: Attribute $P$, target class $C_t$
Ensure: Moving of attribute $P$ from class $C_s$ to class $C_t$
 1: $C_s \leftarrow P.parent$
 2: if $ExistsAnotherAttributeWithSetName(C_t, P.Name)$ then
 3:     // raise a warning
 4:     $raise\ AttributeWithNameAlreadyExistsInClass(P, C_t)$
 5: else
 6:     move attribute $P$ from $C_s$ to $C_t$
 7: end if
 8: $relatedAttributes \leftarrow GetRelationsOf(P, 'AttributeType')$
 9: $relatedtargetClasses \leftarrow GetRelationsOf(C_t, 'ClassType')$
10: for all $rp \in relatedAttributes$ do
11:     for all $rtc \in relatedtargetClasses$ do
12:         if $ExistsAnotherAttributeWithSetName(rtc, rp.Name)$ then
13:             // raise a warning
14:             $raise\ AttributeWithNameAlreadyExistsInClass(rp, rtc)$
15:         else
16:             move attribute $rp$ from $rp.parent$ to $rtc$
17:         end if
18:     end for
19: end for
```

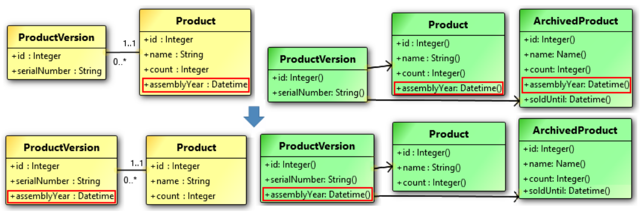

Figure 6: An example of moving attribute *assemblyYear* from class *Product* to class *ProductVersion*

5.7 Cardinalities

Cardinalities are used when there must be defined a (strict) relation between objects. Cardinalities can be expressed over the PIM too (see Figure 2) and they can be used for specific evolution process of the related models. However, cardinalities are not defined in the ReM. Since the model itself follows the tree structure, a cardinality has no significance here.

One situation where the cardinality can be used is during generation of resource addresses from the model because the cardinalities can reduce number of generated addresses when there is relation `[1..1]`.

6. EXPERIMENTS

The presented solution was implemented as an extension of the *DaemonX* framework [9]. *DaemonX* is a plug-inable tool developed as a data and/or process modeling and evolution management framework. All its functionality is provided via various (mainly modeling and evolution) plug-ins which use services provided by the framework. The main services provided by *DaemonX* are integrated environment, support for plug-ins and their inter-operability, and propagation of changes among models created by plug-ins (note that most of the figures in the paper are screen shots of the application). The advantage of this solution is the presence of different plug-ins for different models, e.g., a database model, a model for XML schema, or a model for XPath queries. These models can be incorporated and used together in designing a complex solution.

As the source model we use the existing general *PIM plug-in*. For the purpose of this paper, there were implemented two additional plug-ins. The first one is called *ResourceModel* and represents the model of the REST resource, which is described in Section 4. The second one is an evolution plug-in providing propagation of changes done in the source PIM to the target ReM that implements algorithms described in Section 5.

6.1 Experimental Data

As testing data we used a real-world REST API[1] provided by the GitHub and API[2] provided by Twitter. These APIs were selected because they are well-defined, documented, and commonly used and complex. Data structures used for sending request and especially response messages are complex which is convenient for our experiments too. The model representing a GitHub response consists of 5 classes, 7 associations between these classes, and 48 attributes. There are no functions except for *GET*, *POST*, *PATCH*, and *PUT* methods. For the purpose of the tests, we have added multiple functions with various parameters to be able to test all operations defined over our models.

Figure 7 depicts the PIM used for the experiments and the corresponding ReM after transformation from the JSON definition taken from the GitHub API description. Due to mapping complexity, instead of dashed lines, the mapping is represented with red numbers – the same numbers in PIM and ReM mean that there exists a mapping between these items. As we can see, PIM class *User* is mapped in the ReM to multiple vertices, namely *User*, *Assignee*, and *Creator*. Other PIM classes are mapped to the corresponding ReM vertices. Corresponding attributes are linked with letters.

6.2 Particular Experiments

As we have mentioned, the provided resource API was analyzed and the corresponding PIM and ReM were created. Next, these two models were mutually mapped. And, finally, all PIM operations were applied as described later in this section. Thanks to the implemented algorithms, all changes were propagated to the target ReM and then checked by the following scenario:

1. Make a change in the source PIM.
2. Let the changes propagate to the target ReM.
3. Let the resource API generate from the updated ReM.
4. Check the new resource API.

In particular, the following changes were done in the PIM:

- *Attribute Creating.* Adding of a new attribute to the PIM class will cause creation of a new attribute in the ReM. To be able to perform this operation, all preconditions must be satisfied. We have added new attribute *alias* of type *String* to PIM class *User* (see Figure 7). Since this class is mapped to ReM vertices *User*, *Creator*, and *Assignee*, attribute creation was propagated to these vertices. Since there were no attributes with the same name, name collision did not occur.

[1]https://developer.github.com/v3/pulls/
[2]https://dev.twitter.com/overview/API

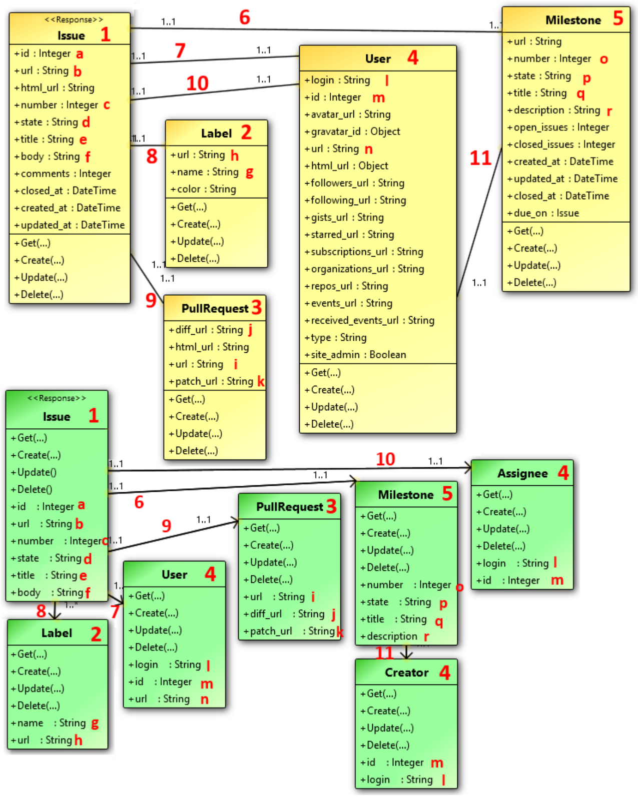

Figure 7: An example of the PIM and its corresponding ReM used for experiments

- *Attribute Removing.* Removing an attribute from a PIM class will cause removal of the related attributes from the ReM. As in the previous case, we focused on PIM class *User*. First we removed attribute *login*. Since this attribute is mapped to attributes in ReM vertices *User*, *Creator*, and *Assignee*, they were removed from these classes too. Next, we removed attribute *type*. Since this attribute has no relations in the ReM, no operation was propagated.

- *Attribute Renaming.* A change of an attribute in the PIM class will cause a change of the respective attributes in the ReM. To be able to perform this operation, preconditions must be satisfied. In this case we renamed attribute *state* of PIM class *Milestone* into *status*. Due to the relation to the attribute *state* in ReM vertex *Milestone* and since there is no attribute with the same name in the class or vertex, this change was successfully propagated. Subsequently we renamed attribute *html_url* in PIM class *User* into name *url*. Since an attribute with the same name already existed, the operation was canceled.

- *Attribute Type Changing.* A change of an attribute data type does not have any impact on the generated resources, because attribute type is not present in the resource. In this case we changed the data type of attribute *state* in class *Milestone* from *String* to enum *MilestoneStateEnum*. Since there is no restriction on the type change, this change was propagated successfully.

- *Attribute Moving.* Moving an attribute from one PIM class to another will cause moving of all related attributes in the ReM. To be able to perform this operation, preconditions must be again satisfied. The algorithm checks all possible collisions of the attribute names. E.g. moving an attribute to a class where there already exists an attribute with the same name or moving multiple attributes to one class represent the same evolution process. Suppose moving of attribute *url* from PIM class *User* to PIM class *Issue*. Since there already exists an attribute with the same name, the operation was canceled. Next, suppose moving of attribute *title* from PIM class *Issue* to PIM class *Label*. Since there is no attribute with the same name in the target class, the operation can be propagated to all related ReM vertices. Again, there exists no attribute *title* in ReM vertex *Label*, the attribute moving can be propagated.

The tests and test results proved that our algorithms were defined correctly and provide expected results. In the presented GitHub model all operations done in PIM class *User* can be propagated to all related ReM vertices i.e. *User*, *Creator*, and *Assignee*. An interested reader may download the implementation as well as the sample models[3].

6.3 DaemonX Extension

Through this paper and the proof-of-concept implementation of the model and evolution plug-ins the DaemonX framework was extended with new functionality. In general, it enables creation of complex models based on the MDA principe, i.e. to have one common PIM and related models for specific platforms, e.g., XML, ER, BPMN, ReM, and even the respective operations, e.g., SQL or XQuery queries.

As depicted in Figure 8 the original version of DaemonX involved three parts – the XML view (the blue part), the storage view (the green part), and the process view (the yellow part). In this paper we have extended the idea with the REST view (the red part). We denote the depicted architecture as five-level evolution management framework. Its strength is in the common gray PIM which describes the problem domain covered by the application. At each level of all the views we have defined the edit operations and respective propagation algorithms, like we have done it in this paper for the REST view. Hence, any change in any part of the system can easily be propagated to all affected parts of all other models. First, it is propagated upwards to the PIM level to identify the affected objects of the modeled reality. Then, we can completely and correctly propagate the modifications downwards to all the affected parts of the system. Note that not all the views involve all the levels or both directions of propagation, because not all of them are necessary from the particular viewpoint of the application. For instance, the process view does not involve the operational level. Or, the upwards propagation from the operational level to the schema level does not make sense in the XML or storage view, because changes in XQuery or SQL queries should not affect the data structures (but of course the opposite direction is very important). More details about the framework can be found in [12, 13].

[3]http://www.ksi.mff.cuni.cz/en/~polak/daemonx/resource-model.zip

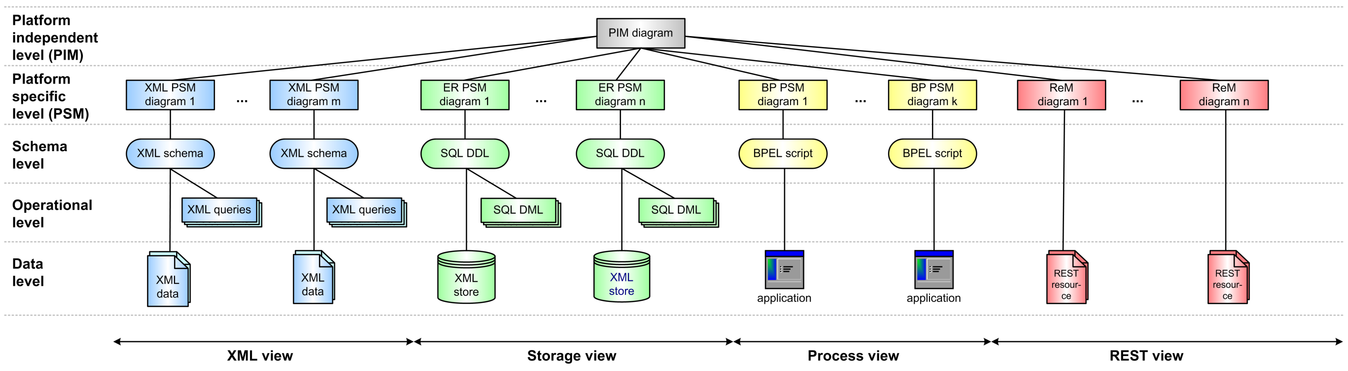

Figure 8: Extended DaemonX architecture

7. CONCLUSION

The main aim and contribution of this paper is to extend our existing solution of maintaining REST resources with a set of new operations over the model and its continuous development during system evolution. The purpose is to reduce the possibility of errors and omittings during manual update and to reduce the amount of work which must be done manually by a developer or an architect. For this purpose we provide algorithms to propagate changes between models based on the MDA approach. As the source model, we use well-known PIM and as the target model we defined new PSM model representing a REST service resource, called Resource Model (ReM). The ReM representing the REST service can be subsequently used for generating a stub for a concrete client and server implementation or a programming language. Next, the model can be combined with existing solutions like [7] or [6] which will then provide a full MDA managed and documented evolution of the REST API. The proposal can also be incorporated together with other PSM models (e.g., database, XML, object etc.) to provide a complex MDA solution. And the proposed algorithms can also be used over existing models – e.g., in IDEs like Visual Studio to provide a similar functionality.

Acknowledgment

The work was supported by the GAUK grant no. 1416213 and project SVV-2015-260222.

8. REFERENCES

[1] API Blueprint. http://apiblueprint.org/.
[2] API Evolution. https://www.mnot.net/blog/2012/12/04/api-evolution.
[3] API Versioning. http://pivotallabs.com/api-versioning/.
[4] Hypertext Application Language. http://stateless.co/hal_specification.html.
[5] JavaScript Object Notation. http://www.json.org/.
[6] RESTful API Modeling Language. http://raml.org/.
[7] Swagger. http://swagger.io/.
[8] C. K. Liu D. Booth. *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*. W3C, June 2007.
[9] DaemonX-Team. DaemonX, June 2011. http://daemonx.codeplex.com.
[10] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000. AAI9980887.
[11] J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. Object Management Group, 2003. http://www.omg.org/docs/omg/03-06-01.pdf.
[12] M. Nečaský, I. Mlýnková (Holubová), J. Klímek, and J. Malý. When Conceptual Model Meets Grammar: A Dual Approach to XML Data Modeling. *International Journal on Data & Knowledge Engineering*, 72:1–30, 2012.
[13] M. Nečaský, J. Klímek, J. Malý, and I. Mlýnková (Holubová). Evolution and Change Management of XML-based Systems. *Journal of Systems and Software*, 85(3):683–707, 2012.
[14] Object Management Group. *UML Superstructure Specification 2.1.2*, November 2007. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/.
[15] M. Polák and I. Holubová. REST API Management and Evolution Using MDA. In *Proceedings of the 8th International C* Conference on Computer Science & Software Engineering*, Yokohoma, Japan, 2015. ACM Press. (In Press).
[16] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures (Second Edition)*. W3C, October 2004. `http://www.w3.org/TR/xmlschema-1/`.
[17] W3C. *Extensible Markup Language (XML)*. W3C, 2010.