UML-based Model-Driven REST API Development

Davide Rossi
Department of Computer Science and Engineering, University of Bologna, Bologna, Italy

Keywords: UML, REST, Web Services, Model-Driven Development.

Abstract: In the last few years we have witnessed the expansion of REST APIs as a method to implement machine-to-machine interactions in open distributed systems. Recently REST APIs can also be found in several B2B and enterprise scenarios that were previously reserved to alternative technologies such as SOAP-based Web Services. Despite that, the development of REST-based solutions has remained mostly inspired by agile approaches with no or limited explicit modeling artifacts produced during the development process. This clashes with software development methods in which modeling artifacts are expected to be available for all developed software. Another problem is related to the resource-based nature of these APIs that miss standardized methods to discover and understand their capabilities akin to what object-oriented interfaces can do for objects and services. In this paper we propose a model-driven approach to REST API development; this approach is composed by two main steps: (i) UML modeling of the API using specific profiles and (ii) a model transformation that exploits RAML, a recent RESTful API modeling language, as an intermediate notation that can be used to automatically produce documentation and code for various languages/platforms.

1 INTRODUCTION

SOAP-based (W3C XML Protocol Working Group, 2007) web services have been the leading integration technology in enterprise and B2B scenarios for the last few years. Besides the obvious advantages related to standardization and interoperability there are arguably two sets of reasons explaining this success. First the availability of a rich technologies stack, enabling the implementation of the advanced non functional requirements enterprise software is usually expected to meet (that includes features such as reliable messaging, advanced security, transactions support and so on). Secondly the ease of integration with existing OO-based modeling methods: while SOAP has been designed to adapt to different interaction models (such as the message-based one) it fits naturally object oriented methods thanks to the natural mapping of WSDL (Web Services Description Working Group, 2007) concepts such as port type and operation to classes and methods. Furthermore, when UML (OMG, 2015b) is adopted in the development process, SoaML (OMG, 2012) provides a OMG-standardized approach for SOAP-based web services modeling within service-based architectures, allowing them to fit nicely in most plan-based methods (such as the large family of UP-inspired (Jacobson et al., 1999) software development processes). All of this, however, comes at a price: complexity. SOAP and WSDL, mostly because of their roots in XML schema, can pose difficulties to newcomers, and the full WS-* stack (as the set of variously supported specifications associated to SOAP-based web services is usually called) has widely been criticized for its intricacy which translates into elaborated software libraries and frameworks to deal with. RESTful API, on the other side, make simplicity their aim. This couples well with the diffusion of agile approaches into the realm of distributed programming and their preference to light, simple, composable technologies and determined a wide adoption of REST-based APIs.

As this adoption started to interest more and more enterprise software projects, the lack of a standardized approach to declare and discover REST API capabilities becomes a relevant issue. What is in fact missing is a uniform way to model available resources, allowable operations, messages parameters and payload formats. This impacts software development: on the provider side no modeling artifact is available to guide the implementation of the API, on the consumer side there is no formal documentation on how to use the API. As a consequence also the concept of a contract among the parties cannot be addressed.

The relevance of this problem can easily be perceived by the number of recent proposals for both

Rossi, D.
UML-based Model-Driven REST API Development.
In *Proceedings of the 12th International Conference on Web Information Systems and Technologies (WEBIST 2016) - Volume 1*, pages 194-201
ISBN: 978-989-758-186-1

the description/modeling of the API itself and for the modeling the data that are exchanged through it (REST API data modeling is arguably one of the main forces behind the development of JSON schema, currently an IETF Internet-Draft[1]). Among the existing proposals we can list WADL[2] (Web Application Description Language), Swagger[3] and RAML[4] (RESTful API Modeling Language). Notice that also WSDL 2.0 can be used to model RESTful services but has been sparingly adopted. Although most of these proposal are a welcome addition, when developing software within a model-first perspective it should be taken into account that they (i) are not visual and (ii) are not UML, that is they are not a standard, widely used and tool-supported notation but are yet another DSL (domain specific language) (Van Deursen et al., 2000) to learn and integrate in the toolchain.

The research work presented in this paper aims at defining a standards-based approach to the model-driven development (Schmidt, 2006) of REST APIs; this approach has been designed under the guidance of the following desiderata (we could have called them requirements, but the idea of self-imposing requirements seems a bit rhetoric), most of them stemming from aforementioned considerations:

D1. The solution should be based on UML: it should be possible to automatically transform an initial UML model into code supporting the implementation of the API and its consumption; automatic or semi-automatic generation of the API documentation is also welcome.

D2. The solution should be agnostic with respect to the language and the platform used to implement the API.

D3. The use of intermediate formats in the D1's transformation is possible but this should not introduce novel DSLs.

D4. The solution should focus on structural modeling; behavioral aspects could be added with future extensions but are not a primary concern at this point.

D5. The solution should not require specific/proprietary software tools; a fully-FOSS software-based tool chain is advisable.

The solution we present here is based on two main contributions:

C1. A flexible REST API modeling approach based on leyered UML profiles.

C2. A model-driven transformation chain based on an existing intermediate representation from which documentation and code for various languages/platform can be refined.

C3. A workflow defining a method to implement the proposed MDE approach.

This paper is structured as follows: section 2 discusses some of the existing proposals for the modeling of RESTful APIs, some of which make use of UML. A brief introduction to RAML can be found here as well. In section 3 our proposal is presented: starting with a set of desiderata we evaluate the possible options and end up with a method making use of UML profiles and model-to-text transformations. In section 4 a typical workflow adopting our approach is presented. Section 5 concludes the paper and outline future work.

2 RELATED WORKS

A few works are related to the approach we present in this paper. In (Rathod et al., 2013) the authors propose an approach based on UML to model structural (via class diagrams) and behavioral (via protocol state machines) aspects of a RESTful API. In the class diagram individual resources are mapped to plain classes whereas collections are explicitly marked using a collection stereotype. No further description of the operations is provided in this diagram: HTTP transactions are used as triggers and state behaviors in the state machine diagram. In a example the authors refine (in a way which is not described) an .NET implementation of the API provider. In (Rauf et al., 2010) the authors propose the use of UML to model the composition of RESTful web services. Structural modeling happens in a way that is very similar to the previously discussed work. Behavioral modeling is also very similar in which state machines are used and the various HTTP invocations are mapped to call behavior actions, the main difference being that in this case what is modeled is the behavior of a composition of services. A conceptual metamodel for RESTful APIs is presented in (Schreier, 2011); also typical resource types (such as primary, sub, list, filter, ...) are taken into account. The author does not map these concepts to any concrete modeling language leaving the option to use them as the basis for a DSL or for a UML extension. In (Ormeno et al., 2012) the authors present a UML profile for modeling RESTful services based on the controllers and artifacts generated by Spring Roo[5]. The (structural) UML model is

[1] http://tools.ietf.org/html/draft-zyp-json-schema-04
[2] https://wadl.java.net/
[3] http://swagger.io/
[4] http://raml.org
[5] http://projects.spring.io/spring-roo/

then transformed, using Enterprise Architect's[6] proprietary Code Template Framework, into a Spring Roo application skeleton. In (Schreibmann, 2014) the authors propose a model-driven approach for the development of RESTful APIs. Their proposal is based on a meta-model that serves as the basis for a DSL used to model the API. By using Xtext[7] the DSL can be converted into Java/JAX-RS. The IBM product Rational Software Architect includes a (non-formally described) REST profile that is described in the documentation library[8]. Stereotyped class diagrams can be transformed to JAX-RS-based Java code using proprietary tools.

As stated in the introduction another aspect related to REST API is the surge of DSLs to document/model them. A comparison between these proposals is outside the scope of this paper, in our work we focus on RAML. This is mostly due to the fact that, as the ML (modeling language) letters in the name implies, this proposal is more oriented to the modeling of the API rather than simply to its documentation/description in favor of potential consumers, enabling what it s called an API design-first approach. RAML is a non-proprietary, vendor-neutral open specification. The workgroup developing the specification includes representatives from MuleSoft, VMware and Cisco. A RAML model is represented in a text document adopting the YAML[9] format (a popular human friendly data serialization standard) and contains a structural description of the API: its resources, their methods, the URI and query parameters used in invocations, the format for the body of the exchanged entities, and so on. The RAML developers and a large user community also make available various tools (usually as open source software), among them several converters to transform RAML specifications in documentation or skeleton code for many language/platform pairs.

3 FROM UML TO REST APIs

The solution we designed is meant to satisfy the desiderata exposed in the introduction section. What follows is a brief discussion of how these requirement affected our choices.

D1 implies that a new modeling notation should not be developed and if a metamodel is defined it should then be possible to express it using UML profiles. This strongly drove us toward the adoption of a UML profile.

D2 implies that a starting model should be easily transformed into different language/platform targets. As a result, no implementation-specific concerns should percolate upward to the starting notation. But this also means that, in order to maximize the adoption of the solution, several transformation targets should be supported (like PHP, Javascript/Node.js, Java/JAX-RS, and so on). Being this far from trivial we started considering the adoption of an intermediate notation even more appealing if that has available code generators that can be adopted in the tool chain.

D3 goes in this very same direction: if the analysis of the other requirements drive toward the adoption of an intermediate notation, this notation should be an existing one, allowing the adoption of existing tools and avoiding the need to learn yet another DSL by the development team members.

Taking into account also D4 and D5 we decided to adopt an intermediate RAML artifact. The decision to use RAML impacts the definition of the initial profile. As usual when adopting transformations in model-driven approaches, the problem of semantic equivalence arises: if the starting notation contains more information than the target, this information is inevitably lost in the transformation. On the other side, if the target notation contains more information then it is possible that the resulting artifact needs to be furtherly modified to add information if that is used in later stages. Since RAML can express implementation details (such as the schema of the payloads or the authentication mechanisms) that can be fruitfully exploited in later transformations and since, by design, we decided that these details should not be part of the UML profile, it is pretty obvious that we are in the second of the two aforementioned cases. What this means is that the RAML file produced by the UML-to-RAML transformation is a skeleton containing structural information and it has to be enriched with additional details before refining it in successive artifacts. The modification of an intermediate artifact in model-driven approaches is always problematic: what happens when the transformation producing the original version of the artifact is re-executed? Should the artifact be overwritten and all the additional elements discarded? For model-to-text transformations most of the times we witness solutions based on the creative use of comments to mark elements that have to be preserved and/or elements that can be overwritten. RAML wisely includes a layering mechanism (based on so called overlays) that allow manual re-

[6] http://www.sparxsystems.com/products/ea/

[7] https://eclipse.org/Xtext/

[8] http://www.ibm.com/developerworks/rational/library/design-implement-restful-web-services/

[9] http://www.yaml.org/

finements of a specification to take place in extension artifacts without the need to modify the initial file produced by the transformation, allowing its regeneration without having to worry about overwriting it. While this is a viable solution for most scenarios we still would like to support development strategies in which adding as much information as possible in the initial UML model is the preferred option. To make this possible we decided to develop a second UML profile, meant to be used as a second layer on top of the first, more general profile. For the first profile we evaluated the use of the existing ones discussed in the related works section but most of them either included a relevant behavioral part or are not fully documented (as is the case for the IBM profile, that we used as inspiration). Given D5 we decided to create the two profiles using the open source Papyrus[10] modeler. The resulting artifacts should be usable in other UML tools with no or minimal effort.

The UML-to-RAML transformation should accept UML models using the generic profile (the REST profile) and use the information added using the second profile (the RAML profile) if present. Since no formal MOF-based meta model exists for RAML we decided to avoid going in the direction of a model-to-model transformation (potentially using QVT (OMG, 2015a) or ATL (Jouault et al., 2006)) targeting the RAML metamodel with a later model-to-text transformation to create the actual RAML YAML file. The transformation is then a model-to-text from UML+REST/RAML profiles to the RAML YAML file.

The architecture (in the sense of significant design decisions that characterize a solution (Buschmann et al., 2007)) we propose is shaped as follows:

- A novel generic REST API profile focusing on structural aspects is developed, this profile is designed so that no implementation details percolates upward.
- A RAML-specific profile is developed, meant to be used along the generic one.
- An UML-to-RAML transformation is defined. The transformation is implemented with the OMG MOF Model to Text Language (MTL) standard (OMG, 2008).
- Existing (open source) tools allowing RAML specifications to be transformed into documentation/code for various languages/platforms are used.
- The profiles and the MTL code is made available as open source software[11].

[10]https://eclipse.org/papyrus/

[11]https://bitbucket.org/DavideRossi/uml2raml

3.1 Two UML Profiles for REST APIs

The two layers (REST only and REST with RAML) approach has been discussed before. This approach could be realized by using different techniques: the RAML profile could be a subprofile of the REST one or they could be totally distinct. We decided to chose for the latter option: in the tool we used subprofiles have to be defined in the same model as the profile they depend on and we wanted to make the REST profile available as fully standalone. Designing a UML profile for REST APIs does pose some technical problem. In a REST API we usually find two kind of elements: single resources and collections. Consider a simple API to access the details for a set of persons. The path to access persons such as Alice, Bob and Cecilia could look like /api/persons/alice, /api/persons/bob, /api/persons/cecilia. That is, the path specification /api/persons/{name} can be used to access all person resources. But also /api/persons indicates a resource, in this case a collection that, for example, can be queried to have the list of all the available persons or be used (possibly using a POST factory pattern (Pautasso, 2014) to create new persons. In an API like this the elements we want to model are the ones accessed via /api/persons/{name} (that represent a class) and the one at /api/persons that represent a single resource (a collection). Mapping these concepts in a OO way would mean to use classes for /api/persons/{name} and instances of a collection type for /api/persons. Not only this is hardly achievable in standard UML even mixing imports and profiles but, most importantly, would result in a very unnatural way of modeling. Some of the existing proposal use distinct stereotypes to mark single resources and collections but this distinction is not really needed from a structural point of view. For this reason we decided for a simpler, yet conceptually not fully accurate, solution: model both concepts with the same stereotyped class. The resulting profile is very straightforward and is represented in figure 1. An example of its application can be seen in figure 2.

The approach is quite straightforward: an API is contained in a API stereotyped package. Resources are modeled as Resource stereotypes classes

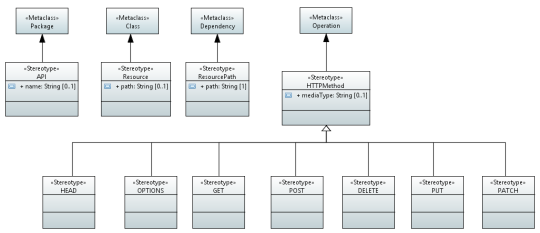

Figure 1: The REST profile as modeled in Papyrus.

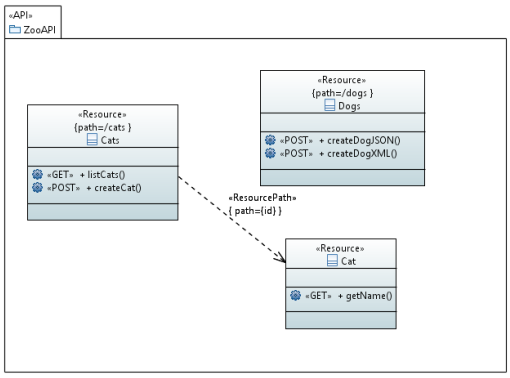

Figure 2: A sample model using the REST profile.

within these packages. These resources could be non-correlated and characterized by their own absolute path or can be explicitly organized in hierarchies by tying them together with a ResourcePath stereotyped dependence. In the latter case the path of a sub-resource is derived from the path of the super-resource it references and from the path property of the ResourcePath stereotype that is applied to the dependency edge.

The second profile contains much more details with respect to the first one, since it has to capture several aspects that can be modeled in RAML (notice that we decided not to represent the whole RAML metamodel). That is, there are valid RAML documents that cannot be represented by the profile. This was never our intention: the profile is meant to help the design of REST APIs via UML and convert them to RAML descriptions adopting the most usual REST API patterns, not to work as a RAML visual notation. If the use of very specific RAML constructs is need, the suggestion is to make use or RAMLs own layering mechanism to enrich the specification produced by the transformation of the UML model. Despite being limited, the RAML profile can easily be misused by trying to model too many details, leading to intricate, unreadable diagrams. Also consider that some of the stereotypes' properties are defined with respect to data types defined in the profile, and the values of these properties cannot be displayed by most existing UML modelers.

The profile, instead, is intended to be used alongside the modularization and reuse mechanisms provided by RAML: data types, resource types and traits. RAML data types can be derived by a set of built-in types (by using scalar specialization, inheritance, arrays, enumerations, unions and maps) or can reference JSON or XML schema. RAML resource types are partial resource definition specifying common characteristics such as security schemes and methods. RAML traits are partial method definitions specifying common characteristics such as description, headers, query string parameters, and responses. So, while is is possible to use the profile to specify that a POST method on a resource accepts a JSON payload composed by a name and a birthDate, the suggested way to proceed is to define a PersonType with these properties in an external file that is imported, and use the profile to specify that the method accepts a PersonType payload. Similarly, while is possible to use stereotypes properties to specify that a collection resource accepts query parameters such as start and count in its GET method, the suggested way to proceed is to define a "paged" trait and refer to that in the UML model. The same applies to resource types.

For terseness we are omitting the other details of the RAML profile here; an example of a model using the two profiles together can be found in figure 3.

Notice the application of stereotypes from both profiles in elements such as in the outer package representing the API. One advantage of this approach is that by filtering out the whole second profile, a valid view of the model with respect to the generic REST profile can be easily obtained.

3.2 From UML to RAML

To convert the UML model adopting one or both the profiles we decided to design a model-to-text transformation using Acceleo[12] an Eclipse-based implementation of the OMG MTL standard; since both Papyrus and Acceleo are part of the Eclipse Modeling Project this should limit interoperability issues. The transformation needs the REST profile to be used; if the RAML profile is also applied the additional information is used. The basic structure of the API is derived from the resources and their hierarchies (specified using the ResourcePath stereotyped dependencies as detailed above). MTL is a transformation language similar to XSLT. Templates are used to define how to transform specific parts of the input UML model (the one matching a pattern specified in OCL); functional expression (based on a superset of the OCL language) are used to manipulate model elements and produce the textual result. The following snippet

```
1  [template public generateAPI
2  (aPackage : Package) ?
3  (hasStereotype(aPackage, 'RestProfile::API'))]
```

shows the definition of a template (generateAPI) to be applied to all the package elements in the source model to which the API stereotype defined in the RestProfile has been applied. hasStereotype is a utility function defined in the same MTL module. Inside

[12]https://www.eclipse.org/acceleo/

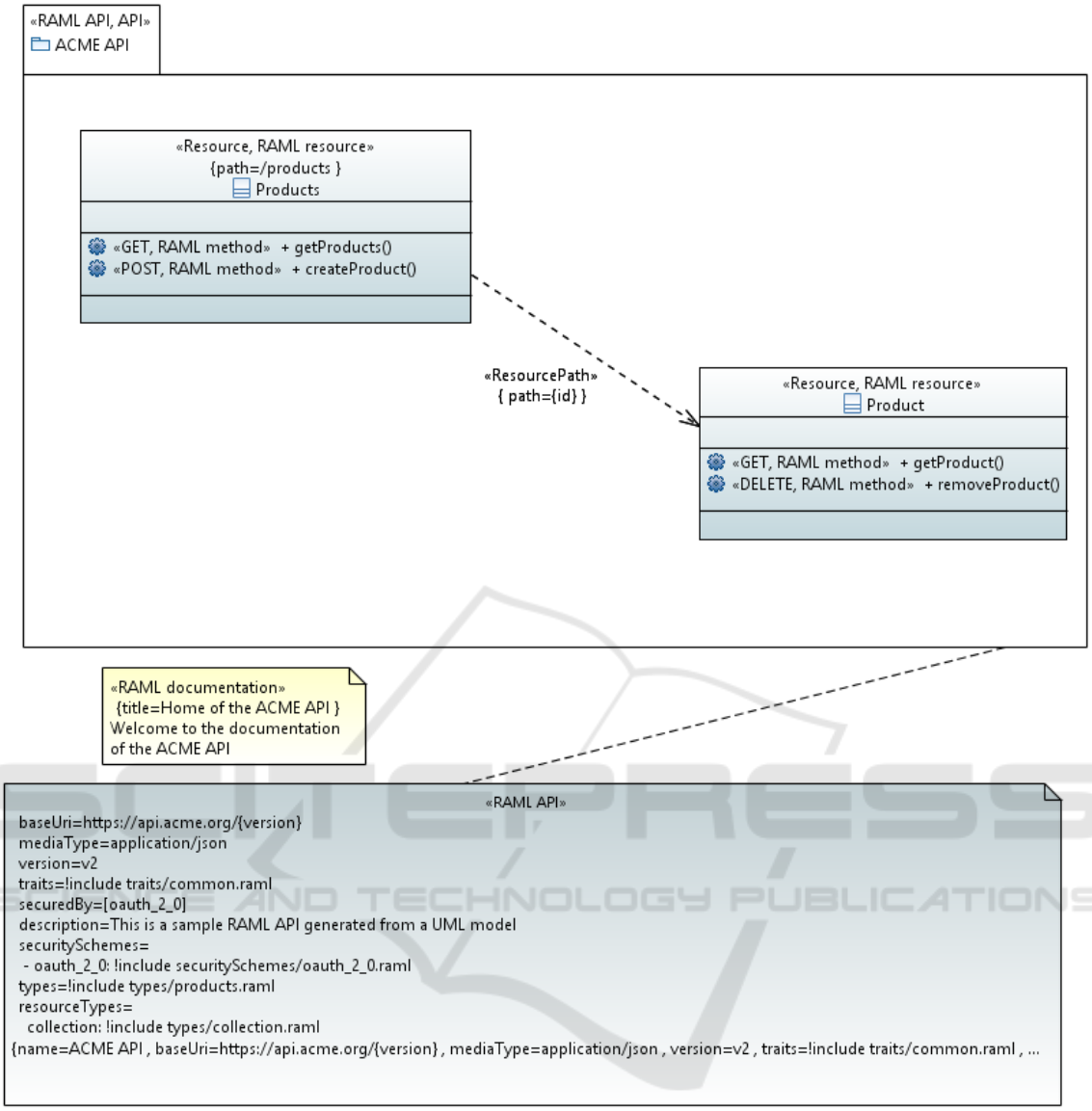

Figure 3: A sample model using both profiles.

the template all the Resource stereotyped classes contained in the package identified by the generateAPI can be iterated by using

```
[for (e : Element | aPackage.ownedElement->
select(e1 | e1.oclIsTypeOf(Class) and
hasStereotype(e1, 'RestProfile::Resource'))]
```

An example fragment of the output of the transformation on the simple model of figure 3 is as follows:

```
#%RAML 1.0
title: ACME API
version: v2
baseUri: https://api.acme.org/{version}
mediaType: application/json
description: |
  This is a sample RAML API generated
  from a UML model
securitySchemes:
 - oauth_2_0: !include security/oauth_2_0.raml
types: !include types/products.raml
resourceTypes:
  collection: !include types/collection.raml
traits: !include traits/common.raml
securedBy: [oauth_2_0]
documentation:
 - title: Home
 - content: |
     Welcome to the ACME API documentation

/products
  displayName: Products
```

```
23    type: collection
24    get:
25      description: Get a list of products
26      is: [paged, secured, rateLimited]
```

4 SUGGESTED WORKFLOW

Having tried this approach with a few realistic test cases we gained some experience and we developed a feel for how to approach the modeling phase. As previously suggested good modeling assumes that the modularization and reuse features of RAML are adopted. Resource types and traits usually respond to patterns an are defined in common libraries and reused among different projects so most of the times no explicit modeling is required (usually some refactoring is needed, though). The situation is a little different for data types: these are usually domain-specific so reuse is not an option. RAML has a well-defined data model and it could be interesting to create a UML profile allowing the definition of RAML types. Many times, however, the RAML data model is ditched in favor of JSON schema and XML schema (something that is fully supported by RAML) so we saw little value in another profile and decided to reconsider it as a potential future work. What follow is the typical workflow we suggest when adopting our approach.

1. A UML model adopting the generic REST profile is created to define the structure of the API (the available resources and their topology).
2. Data is modeled considering all the payloads for all the resources' HTTP methods presented in the model. This is usually realized using either JSON schema or XML schema. If needed RAMLs own type system can be used.
3. Traits common among different methods (such as authentication mechanisms, pagination for resources, REST factory pattern and so forth) are defined. The same is performed with resource types. Both are usually reused from repositories, libraries or other projects.
4. The RAML profile is added to the UML model and stereotypes are used to characterize resources with respect to resource types, methods with respect to traits and payloads and query parameters with respect to data types.

The resulting UML model is clear, readable and, once feeded to the model-to-text transformation, produces a RAML file that can be used for successive transformation with no or minimal editing.

All the activities of this workflow can be realized with open source software tools.

5 CONCLUSIONS AND FUTURE WORK

In this paper we presented a UML-based model-driven approach for the development of RESTful APIs. An initial UML model adopting one or two proposed profiles is created and can be automatically transformed into a RAML file. Depending of the information contained in the UML model, this RAML specification can be directly used to automatically create documentation and code in various languages/platform pairs or can be extended using a layering mechanism before further transformations. Several design alternatives for the method have been discussed and the adopted ones have been chosen to address a set of desiderata stemming from an analysis of the state of the art. As it is usual in these cases, these decisions make sense in the specific context we described, in other contexts other decisions would possibly be preferable. Also notice that we not claim that this is an overall better approach with respect to other existing ones based on DSLs: we are presenting an option for those who value the use of UML in their workflows. Future works on this subject include data modeling and the development of an environment based on patterns and libraries so that a simplified intermediate profile can be used instead of the full RAML one to model in detail most RESTful APIs with minimal or no knowledge of the intermediate notation.

ACKNOWLEDGEMENTS

The research presented in this paper has been partially funded by the Italian grant PRIN in the context the IDEAS project.

REFERENCES

Buschmann, F., Henney, K., and Schimdt, D. (2007). *Pattern-oriented Software Architecture: On Patterns and Pattern Language*, volume 5. John wiley & sons.

Jacobson, I., Booch, G., Rumbaugh, J., Rumbaugh, J., and Booch, G. (1999). *The unified software development process*, volume 1. Addison-wesley Reading.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., and Valduriez, P. (2006). ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN*

symposium on Object-oriented programming systems, languages, and applications, pages 719–720. ACM.

OMG (2008). MOF Model to Text Transformation Languag.

OMG (2012). Service oriented architecture Modeling Language (SoaML).

OMG (2015a). Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT).

OMG (2015b). Unified Modeling Language (UML).

Ormeno, E., Lund, M., Aballay, L., and Aciar, S. (2012). An UML profile for modeling RESTful services. In *13th Argentine Symposium on Software Engineering, ASSE 2012*, pages 119–133.

Pautasso, C. (2014). RESTful web services: principles, patterns, emerging technologies. In *Web Services Foundations*, pages 31–51. Springer.

Rathod, D. M., Parikh, S. M., and Buddhadev, B. (2013). Structural and behavioral modeling of RESTful web service interface using UML. In *Intelligent Systems and Signal Processing (ISSP), 2013 International Conference on*, pages 28–33. IEEE.

Rauf, I., Ruokonen, A., Systa, T., and Porres, I. (2010). Modeling a composite RESTful web service with UML. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 253–260. ACM.

Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):0025–31.

Schreibmann, V. (2014). Design and Implementation of a Model-Driven Approach for RESTful APIs. In *Proc. Fifth IEEE Germany Students Conference*.

Schreier, S. (2011). Modeling restful applications. In *Proceedings of the second international workshop on restful design*, pages 15–21. ACM.

Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-Specific Languages: An Annotated Bibliography. *Sigplan Notices*, 35(6):26–36.

W3C XML Protocol Working Group (2007). Latest SOAP versions.

Web Services Description Working Group (2007). Web Services Description Language (WSDL) 1.1.