WAPIML: Towards a Modeling Infrastructure for Web APIs

Hamza Ed-douibi
IN3 - UOC
Barcelona, Spain
hed-douibi@uoc.edu

Javier Luis Cánovas Izquierdo
IN3 - UOC
Barcelona, Spain
jcanovasi@uoc.edu

Francis Bordeleau
ÉTS
Montreal, Canada
francis.bordeleau@etsmtl.ca

Jordi Cabot
ICREA - UOC
Barcelona, Spain
jordi.cabot@icrea.cat

Abstract—**Web APIs are becoming key assets for any business. Most of these Web APIs are "REST-like", meaning that they adhere partially to the Representational State Transfer (REST) architectural style. The OpenAPI Initiative (OAI) was launched with the objective of creating a vendor neutral, portable, and open specification for describing REST APIs. The initiative has succeeded in attracting major companies and the OpenAPI specification has become *de facto* format for describing REST APIs. However, there is currently a lack of tools to provide modeling facilities for developers who want to manage and visualize their OpenAPI definitions as models and integrate them into model-based processes. In this paper, we propose WAPIML an OpenAPI round-trip tool that leverages model-driven techniques to create, visualize, manage, and generate OpenAPI definitions. WAPIML embeds an OpenAPI metamodel but also an OpenAPI UML profile to enable working with Web APIs in any UML-compatible modeling tool.**

Index Terms—**UML, UML Profile, REST APIs, OpenAPI**

I. Introduction

Consuming, developing and evolving Web APIs are common tasks in many companies' daily activities [1]. This is also reflected by the growing number of available public Web APIs, listed in catalogs such as API Harmony[1] (over 1,000 APIs), RapidAPI[2] marketplace (over 8,000 APIs), APIs.guru[3] (over 800 APIs), or ProgrammableWeb (over 21,000 APIs). In practice, most of these Web APIs are "REST-like", meaning that they partially adhere to the Representational State Transfer (REST) architectural style. REST is a technical description which outlines the principles, properties, and constraints to build Internet-scale distributed hypermedia systems. Due to its lightweight nature, adaptability to the Web and scaling capability, REST has become the preferred style for building Web APIs.

However, REST is a design paradigm and does not propose any standards to describe REST APIs. This situation triggered the creation of several specification languages and protocols to describe REST APIs (e.g., Swagger[4], API Blueprint[5], RAML[6]) and design them (e.g., OData[7]), which makes choosing one format or another subjective to API providers.

To face this situation, a consortium of major actors in the API market has launched the OpenAPI Initiative (OAI) with the goal of standardizing the way to describe REST APIs. OAI has succeeded in attracting major companies in the API ecosystem, including its competitors (e.g., MuleSoft[8], the creator of RAML; or Apiary[9], creator of API Blueprint). As a result, the OpenAPI specification (formally Swagger specification) is now *de facto* format for describing REST APIs.

Despite the growing importance of Web APIs, and OpenAPI in particular, there is a lack of tools to provide modeling facilities for developers who want to follow any kind of model-based approach for the design of OpenAPI definitions. To overcome this situation, we present in this paper WAPIML, an OpenAPI round-trip tool that leverages model-driven techniques to create, visualize, manage, and generate OpenAPI definitions. Once Web APIs are represented as models we can easily integrate them with the rest of modeling efforts in the company, facilitating a global model-driven engineering approach including not only such model representations of the APIs but also their interactions with the rest of the system.

Using WAPIML, developers can import OpenAPI definitions (or create them from scratch), edit the generated class diagrams with any UML Eclipse editor (e.g., Papyrus) and export them to generate the corresponding OpenAPI definition.

Our final goal is to advance towards the definition of a Model-Driven Engineering (MDE) infrastructure for Web APIs with a specific focus on current APIs specifications (e.g., the OpenAPI specification) and protocols (OData protocol), thus facilitating the integration of Web APIs in all kinds of model-based processes.

The remainder of this paper is structured as follows. Section II provides a global overview of WAPIML and its main components. Section III presents the OpenAPI metamodel and UML profile that are at the core of the tool. Section IV discusses the tool support. Section V presents related work. Finally, Section VI concludes the paper and presents some future work.

[1]https://apiharmony-open.mybluemix.net/public
[2]https://rapidapi.com/
[3]https://apis.guru/openapi-directory/
[4]http://swagger.io
[5]https://apiblueprint.org/
[6]https://raml.org/
[7]https://www.odata.org/
[8]https://swagger.io/blog/news/mulesoft-joins-the-openapi-initiative/
[9]https://blog.apiary.io/We-ve-got-Swagger

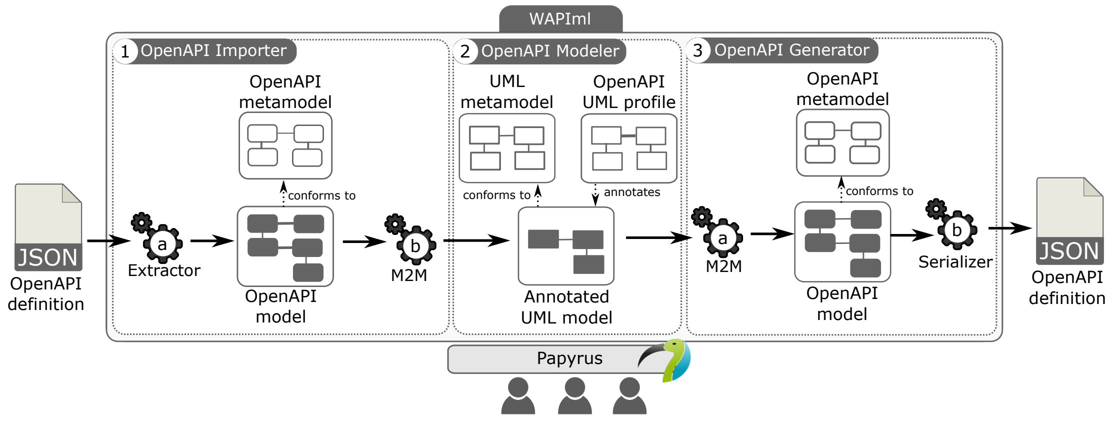


Fig. 1. Overview of the WAPIml tool.

II. WAPIml Overview

WAPIml is an Eclipse-based editor for OpenAPI. It includes both an OpenAPI metamodel and a UML-based version of such metamodel, in the form of a UML profile. Beyond purely modeling a Web API, WAPIml supports the reverse engineering of existing API specifications (from its OpenAPI definitions) and the code-generation of OpenAPI files from our OpenAPI models (or UML-annotated ones).

Our approach is depicted in Figure 1. As can be seen, it is composed of three components: (1) OpenAPI Importer, (2) OpenAPI Modeler and (3) OpenAPI Generator; all of them relying on the OpenAPI metamodel and the OpenAPI UML profile.

The metamodel provides the foundation for the reverse and forward engineering activities as it allows capturing all of the elements of an OpenAPI definition in a model representation. The profile allows reusing existing UML tooling infrastructure (i.e., UML2[10] and Papyrus[11]) to create an editor for OpenAPI; and also promotes integration with other model-based processes. The metamodel and the profile will be presented in more detail in Sections III-A and III-B, respectively.

OpenAPI Importer is in charge of generating a UML class diagram annotated with OpenAPI stereotypes from an OpenAPI definition, and relies on two subprocesses. The first subprocess (see process 1.a in Figure 1) extracts a model conforming to our OpenAPI metamodel from the input OpenAPI definition. This process is almost straightforward since our OpenAPI metamodel mimics the structure of the OpenAPI specification. The second subprocess (see process 1.b in Figure 1) performs a model-to-model transformation to generate a model conforming to the UML metamodel annotated with the OpenAPI UML profile from the previously extracted OpenAPI model. This transformation iterates over the operations and definitions of the OpenAPI model in order to generate classes, properties, operations, data types, enumeration, and parameters, accordingly. These elements are enriched with OpenAPI stereotypes to complete their definitions. This process relies on a set of heuristics to identify the most adequate UML class to attach each OpenAPI operation to. The full list of heuristics can be found in the tool repository[12].

OpenAPI Modeler provides an editor for OpenAPI definitions based on the UML metamodel and the OpenAPI UML profile. The profile can be used to annotate both new or existing UML class diagrams. The editor relies on Papyrus modeling environment.

OpenAPI Generator is in charge of generating an OpenAPI definition from a UML class diagram annotated with the OpenAPI UML profile. It applies a process similar to the importer but in the reverse order. The first subprocess (see process 3.a in Figure 1) performs a model-to-model transformation to generate a model conforming to our OpenAPI model and then the second subprocess (see process 3.2 in Figure 1) performs a model-to-text transformation to generate an OpenAPI-compliant JSON file.

III. Modeling REST APIs

This section describes the modeling artifacts we created to support the OpenAPI specification at the model level, namely: the OpenAPI metamodel and the OpenAPI UML profile.

A. The OpenAPI Metamodel

The OpenAPI metamodel is derived from the concepts and properties described in the OpenAPI specification[13]. Figure 2 shows an excerpt of a simplified version of this metamodel. The `API` element is the root of the metamodel and includes the main attributes to specify the API, like the host or its base path, among others. It also includes references to the data types (i.e., `Schema` elements) used by the operations (`definitions` reference) and the paths of the API (`paths` reference).

The `Schema` element defines the data types that can be consumed and produced by operations. It uses a subset of the JSON Schema Specification defined in the superclass

[10]https://www.eclipse.org/modeling/mdt/?project=uml2
[11]https://www.eclipse.org/papyrus/

[12]https://github.com/opendata-for-all/WAPIml\#notes
[13]https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md

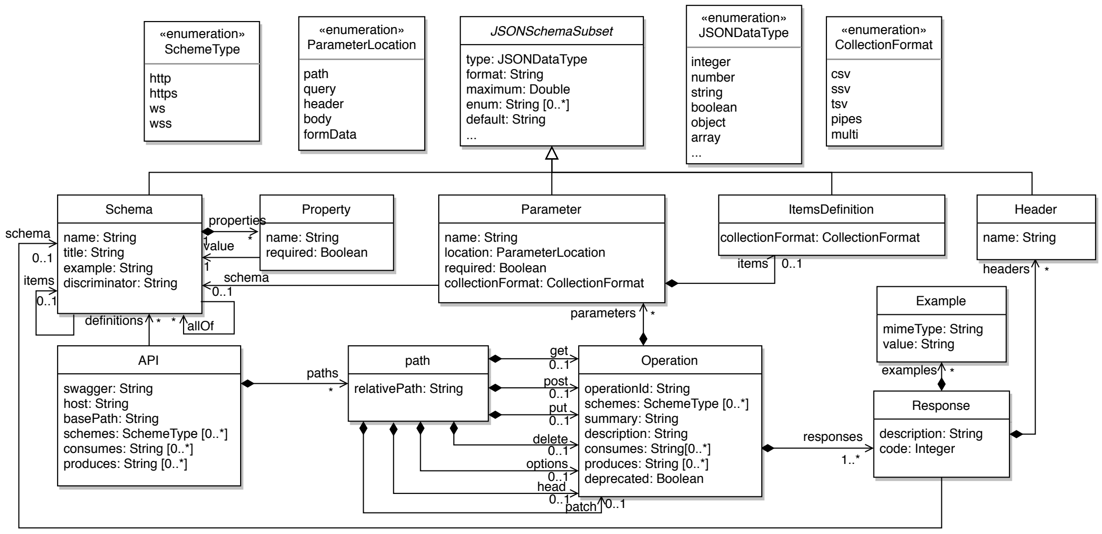


Fig. 2. Excerpt of the simplified OpenAPI metamodel.

JSONSchemaSubset to define constraints. Inheritance and polymorphism are specified by using the allOf reference and the discriminator attribute, respectively. The items reference is used to specify the items of array schema elements.

The Path element defines a relative path of an API endpoint and the supported operations (e.g., get reference for GET HTTP method). The Operation element defines an API operation and includes its parameters (parameters reference) and responses (responses reference).

The Parameter element defines an operation parameter and includes attributes for its name and its location (e.g., see values of ParameterLocation), among others. It inherits from JSONSchemaSubset to define constraints. When the parameter location is body, the schema reference is used to specify its data type. When the parameter location is array, the items reference and the collectionFormat attribute specify its items and the collection format, respectively.

The Response element defines the possible responses of an operation and includes the response code and the response headers, among others. When the response returns data, the schema reference defines its data type.

The complete OpenAPI metamodel, comprised of 31 different metaclasses, is available at our GitHub repository[14] and covers all the aspects of the OpenAPI specification (e.g., security elements or metadata information).

B. The OpenAPI UML profile

We created a UML profile to extend standard UML class diagrams with support for OpenAPI definitions. This enables OpenAPI designers to keep using their preferred UML modeling tool while still benefiting from our work and it also facilitates the integration of our model-driven OpenAPI approach in companies following a UML-based development process.

TABLE I
MAPPING OPENAPI AND UML ELEMENTS.

OPENAPI ELEMENT	CONDITION	UML ELEMENT	DETAILS
Schema	A Schema definition of object	Class	-
Property	A Schema property of type primitive or array of primitives	Property	A class attribute
	A Schema property of type object or array of objects	Property	Association end
Operation	-	Operation	-
Parameter	-	Parameter	direction = in
Response	-	Parameter	direction = return

We will explain first how we map OpenAPI elements to UML concepts and then illustrate some stereotypes created for OpenAPI definitions.

1) Mapping UML and OpenAPI: We use UML class diagrams as the basis of our OpenAPI UML profile due to its semantic similarities with the OpenAPI specification. With UML class diagrams we represent the data model (i.e., schema definitions) and operations of OpenAPI definitions.

Table I shows how we map OpenAPI elements to UML concepts. Columns one and two show an OpenAPI element and a condition (if any), respectively; while columns three and four show the corresponding UML element and the details of such element, respectively. Some mappings are more complex than others depending on the semantic gap between UML and OpenAPI. For instance discriminator and allOf properties (used to define hierarchy in OpenAPI) are represented by inheritance in a class diagram. The complete list of rules can be found in our repository[15].

2) The OpenAPI UML Profile: Based on the previous mappings, we define the profile that includes a set of stereotypes,

[14]https://github.com/opendata-for-all/openapi-metamodel

[15]https://github.com/opendata-for-all/openapi-profile

properties and data types required to complete the modeling of OpenAPI definitions in UML class diagrams. API definitions are represented by `Model` elements of UML class diagrams. We defined the `API` stereotype, which extends the metaclass `Model` and includes the global details of an API.

`Schema`, `APIProperty`, `APIDataType` stereotypes extend the UML metamodel (`Class`, `Property` and `PrimitiveType` metaclasses) to support data types according to OpenAPI, in particular, to define schema objects, schema properties and primitive types, respectively.

The stereotype `APIOperation` extends the metaclass `Operation` to define API operations. It includes information such as the relative path of the operation or the HTTP method of the operation, among others. Note that the concept *path* is not present in the OpenAPI UML profile. Thus, the information of the path and the appropriate HTTP method are included in the definition of the `APIOperation` stereotype.

The stereotypes `APIParameter` and `APIResponse` extend the metaclass `Parameter` to define operation parameters and responses, respectively. `APIParameter` includes information such as the parameter location and the collection format (for multivalued parameters). It also extends `JSONSchema` stereotype to add additional JSON schema constraints. The stereotype `APIResponse` includes information such as the HTTP status code or the list of response headers. Note that these stereotypes do not include information about the data types associated with the parameter or the response, as such information is defined by the type of the UML parameter.

The complete profile, which contains 13 stereotypes, supports all aspects of the OpenAPI specification (e.g., metadata, security) and can be found in our GitHub repository.

IV. Tool Support

WAPIml is available as a set of plugins for Eclipse[16] extending the UI of the platform to provide: (i) two contextual menus to generate either a simple UML model or a UML model annotated with OpenAPI UML profile, from an OpenAPI definition; (ii) an editor for these models provided as an extension to UML2 plugin[17] (i.e., tree-based format plus properties view) and Papyrus[18] modeling environment (i.e., diagram-based format plus rich properties view); and (iii) a contextual menu to generate an OpenAPI definition in JSON from a UML model annotated with OpenAPI UML profile.

Figure 3 shows a screenshot of our tool illustrating an excerpt of the generated class digram (annotated with OpenAPI UML profile) from the *Petstore* OpenAPI definition[19], displayed using Papyrus. As can be seen, the `Pet` class, which represents a pet definition, is annotated with the stereotype `schema`, while its extracted attributes are annotated with `APIProperty`, and its extracted operations are annotated with `APIOperation` and `Security`. The tag values complementing the applied stereotypes can be showed using comments (e.g., see the `Schema` comment box next to the class `Pet`) and modified using the *Properties* view (e.g., see the lower part of the screenshot showing the stereotype `APIOperation` of the `findPetsByStatus` operation).

[16]https://github.com/opendata-for-all/wapiml
[17]https://www.eclipse.org/modeling/mdt/?project=uml2#uml2
[18]https://www.eclipse.org/papyrus/
[19]https://petstore.swagger.io/v2/swagger.json

V. Related Work

The OpenAPI specification is supported by a number of tools providing features such as the generation of SDKs (e.g., OpenAPI Generator[20]), the generation of documentation pages (e.g., ReDoc[21] and Swagger UI[22]), and the manual creation of OpenAPI definitions with a number of IDEs and editors (e.g., Swagger Editor[23], KaiZen OpenAPI Editor[24] or the Senya Editor[25]). Nevertheless, none of these tools provide a graphical visualization of the OpenAPI operations and structure.

JSONDiscoverer [2] allows visualizing the data schema of JSON-based REST APIs while focusing on the inputs/outputs of the operations. However, it does not model the operations themselves nor supports OpenAPI descriptions. OpenAPItoUML [3] is the predecessor of WAPIml. It proposed the generation of UML models to describe OpenAPI definitions. WAPIml extends it by automatically applying the profile to imported OpenAPI definitions and by offering a code-generation option to recreate the (modified) OpenAPI definition.

Only a few other tools have some kind of graphical / modeling editors for OpenAPI. RepreZen API Studio[26], Stoplight[27] and Visual API designer[28] are the best examples. Nevertheless, they all provide their own adhoc editors and notation which hampers their integration with any other modeling environment (e.g. no support for EMF nor for any kind of UML tool or UML-like syntax is provided).

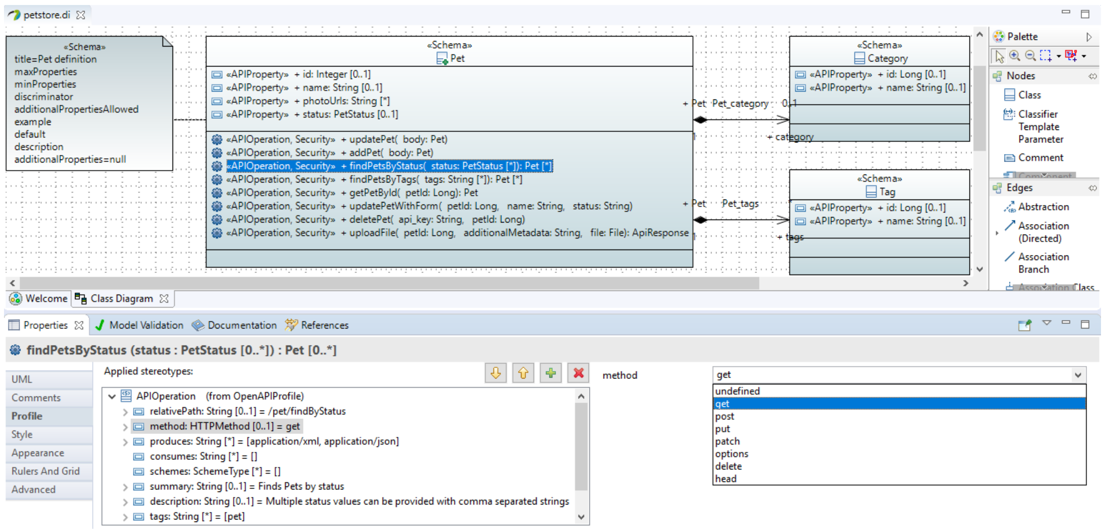
WAPIml complements other MDE techniques used to automate different development tasks for Web APIs in general (e.g., [4], [5], [6], [7], [8], [9], [10], [11]) by potentially enhancing them with specific OpenAPI import, modeling and generation support.

VI. Conclusion

We have presented WAPIml, an OpenAPI round-trip tool that leverages model-driven techniques to create, visualize, manage, and generate OpenAPI definitions using an OpenAPI DSL also expressed as a UML profile for a simpler integration with existing modeling tools. WAPIml has been implemented as a set of Eclipse plugins.

We believe this tool will help modelers who would like to model their Web APIs prior to its actual implementation and will enable the integration of such modeled APIs within the rest of the model-based engineering processes already in

[20]https://openapi-generator.tech/
[21]https://redocly.github.io/redoc/
[22]https://swagger.io/swagger-ui/
[23]https://editor.swagger.io
[24]https://github.com/RepreZen/KaiZen-OpenAPI-Editor
[25]https://senya.io/
[26]https://www.reprezen.com/
[27]https://stoplight.io/
[28]https://www.visual-paradigm.com/features/visual-api-designer/



«Schema»
title=Pet definition
maxProperties
minProperties
discriminator
additionalPropertiesAllowed
example
default
description
additionalProperties=null

«Schema»
Pet
«APIProperty» + id: Integer [0..1]
«APIProperty» + name: String [0..1]
«APIProperty» + photoUrls: String [*]
«APIProperty» + status: PetStatus [0..1]
«APIOperation, Security» + updatePet(body: Pet)
«APIOperation, Security» + addPet(body: Pet)
«APIOperation, Security» + findPetsByStatus(status: PetStatus [*]): Pet [*]
«APIOperation, Security» + findPetsByTags(tags: String [*]): Pet [*]
«APIOperation, Security» + getPetById(petId: Long): Pet
«APIOperation, Security» + updatePetWithForm(petId: Long, name: String, status: String)
«APIOperation, Security» + deletePet(api_key: String, petId: Long)
«APIOperation, Security» + uploadFile(petId: Long, additionalMetadata: String, file: File): ApiResponse

+ Pet Pet_category 0..1 + category
+ Pet Pet_tags + tags

«Schema»
Category
«APIProperty» + id: Long [0..1]
«APIProperty» + name: String [0..1]

«Schema»
Tag
«APIProperty» + id: Long [0..1]
«APIProperty» + name: String [0..1]

findPetsByStatus (status : PetStatus [0..*]) : Pet [0..*]

Applied stereotypes:
APIOperation (from OpenAPIProfile)
relativePath: String [0..1] = /pet/findByStatus
method: HTTPMethod [0..1] = get
produces: String [*] = [application/xml, application/json]
consumes: String [*] = []
schemes: SchemeType [*] = []
summary: String [0..1] = Finds Pets by status
description: String [0..1] = Multiple status values can be provided with comma separated strings
tags: String [*] = [pet]

method: get
undefined
get
post
put
patch
options
delete
head

Fig. 3. A screenshot of the WAPIML tool.

place in the companies. Indeed, WAPIML brings to the table the power of MDE in the context of Web APIs. API models derived using WAPIML could be used to automate different development tasks related to Web APIs such as testing, generation of clients/server SDKs and modeling additional API aspects (e.g., dynamic behavior) by using other UML diagrams, among others. WAPIML has been made available at GitHub[29]. A video illustrating the tool is also available[30].

One of our next steps is to work on the systematic validation of the WAPIML tool. For this purpose, the first phase will consist in using a set of existing APIs to validate that we can import existing OpenAPI definitions and generate a semantically equivalent OpenAPI definition from the unchanged model. We will also need to validate that any modification done at the model level is correctly reflected in OpenAPI definition. Additionally, we plan to add support for validating OpenAPI definitions at the model level (e.g., using OCL), thus allowing modelers to correct their definitions as they model them. Moreover, we will also add a set of heuristics to infer more information from OpenAPI definitions, especially regarding associations between concepts. OpenAPI does not have explicit support for references between types but these could be somehow deduced from the attribute name and type in the JSON schema in order to generate richer UML diagrams.

OpenAPI is just the first standard to be supported by WAPIML. In the near future, we plan to extend WAPIML and transform it into a real Web API modeling platform. Examples of other types of APIs to be added to WAPIML will be OData (partially done already, see our previous work [12], [13]), Socrata [31] (often used in Open Data scenarios), and OpenAPI v3.0, which may allow us to generate other types of UML diagrams (e.g., sequence diagram for operation execution order).

[29]https://github.com/opendata-for-all/wapiml
[30]http://hdl.handle.net/20.500.12004/1/C/MODELS/2019/212
[31]https://dev.socrata.com/

ACKNOWLEDGMENT

This work has been supported by the Spanish government (TIN2016-75944-R project).

REFERENCES

[1] M. Bortenschlanger and S. Willmott, "The API Owner's Manual," 3Scale, Tech. Rep., 2014.
[2] J. Cánovas Izquierdo and J. Cabot, "JSONDiscoverer: Visualizing the schema lurking behind JSON documents," *Knowl.-Based Syst.*, vol. 103, pp. 52–55, 2016.
[3] H. Ed-douibi, J. L. Cánovas Izquierdo, and J. Cabot, "OpenAPItoUML: A Tool to Generate UML Models from OpenAPI Definitions," in *Int. Conf. on Web Engineering*.
[4] H. Ed-Douibi, J. L. Cánovas Izquierdo, A. Gómez, M. Tisi, and J. Cabot, "EMF-REST: Generation of RESTful APIs from Models," in *Symp. On Applied Computing*, 2016, pp. 1446–1453.
[5] A. M. Segura, J. S. Cuadrado, and J. de Lara, "ODaaS: Towards the Model-driven Engineering of Open Data applications as Data Services," in *Int. Conf. on Enterprise Distributed Object Computing, Workshops and Demonstrations*, 2014, pp. 335–339.
[6] F. Haupt, D. Karastoyanova, F. Leymann, and B. Schroth, "A Model-Driven Approach for REST Compliant Services," in *Int. Conf. on Web Services*, 2014, pp. 129–136.
[7] J. M. Rivero, S. Heil, J. Grigera, E. Robles Luna, and M. Gaedke, "An Extensible, Model-driven and End-User Centric Approach for API Building," in *Int. Conf. on Web Engineering*, S. Casteleyn, G. Rossi, and M. Winckler, Eds., 2014, pp. 494–497.
[8] R. Rodríguez-Echeverría, F. Macías, V. M. Pavón, J. M. Conejero, and F. Sánchez-Figueroa, "Model-driven Generation of a REST API from a Legacy Web Application," in *Int. Conf. on Web Engineering, Workshops*, 2013, pp. 133–147.
[9] I. Porres and I. Rauf, "Modeling Behavioral RESTful Web Service Interfaces in UML," in *Symp. on Applied Computing*, 2011, pp. 1598–1605.
[10] N. A. Tavares and S. Vale, "A Model Driven Approach for the Development of Semantic RESTful Web Services," in *Int. Conf. on Information Integration and Web-based Applications & Services*, 2013, p. 290.
[11] J. M. Rivero, S. Heil, J. Grigera, M. Gaedke, and G. Rossi, "MockAPI: an Agile Approach Supporting API-first Web Application Development," in *Int. Conf. on Web Engineering*, 2013, pp. 7–21.
[12] H. Ed-douibi, J. L. Cánovas Izquierdo, and J. Cabot, "A UML Profile for OData Web APIs," in *Int. Conf. on Web Engineering*, 2017, pp. 420–428.
[13] ——, "Model-driven Development of OData Services: An Application to Relational Databases," in *Int. Conf. on Research Challenges in Information Science*, 2018, pp. 1–12.