Enriching API Descriptions by Adding API Profiles Through Semantic Annotation

Meherun Nesa Lucky, Marco Cremaschi(✉), Barbara Lodigiani, Antonio Menolascina, and Flavio De Paoli

University of Milan - Bicocca, Viale Sarca 336, Milan, Italy
{meherun.lucky,cremaschi,depaoli}@disco.unimib.it,
{b.lodigiani,a.menolascina}@campus.unimib.it

Abstract. In recent years several description tools and formats have been introduced for describing REST Web APIs both in human and machine readable formats. Although these descriptions provide functional information about the APIs (e.g. HTTP methods, URIs, model schema, etc.), the information that qualifies the properties of APIs (e.g. classification of input arguments and response data) is missing. We envisage that providing a complete set of information to the users will facilitate the composition of APIs to fulfil users' specific needs.

This paper analyses the current state of the art in Web API Descriptions and Semantic Annotations to show that although there are solutions with semantic capabilities, most of them fails to add semantic annotations automatically or semi-automatically. Moreover, advanced technical skills are needed to manage semantics and compose different Web APIs, which reduce the number of potential users of such solutions. The goal is to enhance actual API descriptions by creating a simple description format to annotate properties at semantic level to support semi-automatic composition. To achieve this goal, we propose an extension of the Open API Initiative (OAI) specification to create comprehensive descriptions. The approach focuses on the emerging concept of API Profiling to add descriptive information of data semantics by addressing Dublin Core Application Profile (DCAP) guidelines.

1 Introduction

As web-enabled software becomes the standard for business processes, the ways organisations, partners and customers interface with it have become a critical differentiator on the market. Therefore, the ability to provide appropriate and complete Web API descriptions to let users discover applications that satisfy a set of requirements and compose applications to fulfil more complex users' needs is critical for the success of any organisation. Although the process of implementing APIs has become common practice, meta-level API definition and implementation have yet to be settled to set widely-accepted standards. Today, description formats, such as Open API Initiative (OAI) specification[1], also known as

[1] http://openapis.org/specification.

© Springer International Publishing Switzerland 2016
Q.Z. Sheng et al. (Eds.): ICSOC 2016, LNCS 9936, pp. 780–794, 2016.
DOI: 10.1007/978-3-319-46295-0_55

Swagger[2], RAML, API Blueprint[3], are available to describe implementation details including resources, access points, status codes and input arguments [7]. These description formats are created by following the API-first approach[4] and using a meta-language based on XML, JSON or YAML. Moreover, a set of tools have been developed to create API descriptions interactively: such tools can auto-generate server-side code, testing options for different HTTP methods, or even fully functional API Clients (e.g. Swagger Codegen[5]). These formats and tools are mostly human-driven and lack supports for detailed information that qualifies the properties of an API (e.g. classification of input arguments and response data). Moreover, these formats may meet the requirements of developers to complete simple tasks, but they are inefficient in advanced API discovery or API composition due to the lack of machine processable semantics [19]. Moreover, such formats should be made easy to understand when the target users include high-level business experts or specific groups of people (e.g. the elderly, people with disabilities, etc.) who do not have specific programming expertise. We name these users as "*end-user developers*". Several studies [1,8] show the need of interactive documentation that provides flexible navigation alternatives with a comprehensive set of information to support a wide range of users. As users background influences how they navigate the documentation, there are barriers for *end-user developers*, due to inconsistent and very technical terminology use. The final goal of our work is to develop descriptions that can be (semi) automatically composed by developers to support end-user composition of APIs, therefore we address the following questions:

- Are there widely adopted approaches, tools or standards for creating machine processable API descriptions with semantics?
- What are the missing features in current API Description formats to aid composition?
- How can existing approaches be improved by adding semantics to API Descriptions to facilitate (semi) automatic user-driven composition?

In this work, we consider the composition of REST Web APIs by adding machine readable semantic descriptions. The approach is to describe properties with semantic meaning by linking to concepts in shared vocabularies.

In the real world, if developers want to compose APIs, they may search directories such as Programmable Web[6], and understand the meaning of involved data, e.g. that *address* means *city* and *street*, or *latitude* and *longitude*, but a machine agent is unable to understand the meaning without a shared representation of property semantics. The use of links to concepts in shared vocabularies that allow a machine agent to compare and compose the actual data can address this issue. We propose to exploit API profiles to provide descriptive information

[2] https://www.swagger.io.
[3] http://raml.org/, https://apiblueprint.org/.
[4] http://www.api-first.com/.
[5] http://swagger.io/swagger-codegen/.
[6] http://www.programmableweb.com.

about the contents of the response according to the Application Profile[7] approach described as a set of metadata elements, policies, and guidelines defined for a particular application.

In this paper we evaluate the current approaches to create API descriptions and make a proposal to include additional qualifying information. Our goal is to enhance interoperability and composition by creating a standard description format that correlate properties at semantic level. To achieve this goal, we propose to include API profiles with the API descriptions created by following the OAI specification.

We adopt the Dublin Core Application Profile (DCAP) guidelines[8], to share data semantics in a specific representation format. We propose the use of a (semi) automatic method for adding annotation, TableMiner [24], which is a semantic table interpretation method to extract the most appropriate concepts from shared vocabularies in a (semi) automatic way by using context information. We will explain this technique elaborately in Sect. 2.

We conceived our approach to use existing vocabularies (about 558) indexed in the Linked Open Vocabularies search engine[9]. The statistics presented in [10] and [18] shows that the most used vocabularies are not domain dependent and as such they may cover different topical categories such as media, government, publications, life sciences, geographic, cross-domain, user-generated content, and social networking. However, the above vocabularies may not provide all needed terms thus, we can rely on additional domain specific vocabularies to get a practical solution covering a large set of areas. They will be integrated with existing vocabularies to ensure practicability.

The rest of the paper is organized as follows, Sect. 2 discusses the state of the art and motivation, Sects. 3 and 4 discusses our proposal, Sect. 5 describes how the system works and Sect. 6 drives some conclusions.

2 State of the Art and Motivation

Although there are many approaches proposed to enrich Semantic Web, each one claiming to be better than the others, strict methodologies to compare the existing description techniques and scientific evidences are missing [20]. In one hand, there are many works that have been proposed in Semantic Web Service community with concrete implementation, on the other hand they lack in facilitating automation in reality due to: (i) the manual work required to create descriptions and, (ii) the lack of standard that limits interoperability. To analyse the current state of the art aiming to facilitate user-driven API composition, we discuss existing approaches that facilitate the use of Web APIs by machine agents. We also analyse the existing approaches and tools considering API Descriptions, Semantic Correlation and Composition.

[7] http://dublincore.org/documents/2001/04/12/usageguide/glossary.shtml.
[8] http://dublincore.zsaorg/documents/profile-guidelines/.
[9] http://lov.okfn.org/dataset/lov/.

To accelerate the use of Web APIs by machine agents, Semantic-Web researchers proposed a number of solutions. Paper [20] emphasises on the need to provide self-descriptive descriptions that include metadata, that can be interpreted by machine agents in a bottom up way (i.e. information structure should be in pieces to whole). Paper [9] proposes a set of best practices to build self-descriptive RESTful services accessible by both humans and machines. Moreover, it defines a framework that extracts compliant descriptions from documents published on the Web, and makes them available to clients as resources. Paper [5] develops Hydra, a small vocabulary to describe Web APIs; this approach aims to introduce a new breed of interoperable Web APIs by breaking the descriptions down into small independent fragments. Paper [6] focuses on facilitating composition process by reasoning with tailored ontologies that capture user preferences.

To analyse current approaches regarding Descriptions, our discussion includes WSDL, WADL, hREST, RDFa, MicroWSMO, SA-REST, MSM, RESTdesc, SEREDASj following the discussions in [4,12,19,22]. Also metadata formats like Swagger, RAML, Hydra and API Blueprint have been discussed following papers [5,17]. WADL is specifically used for syntactic descriptions of RESTful services, instead of WSDL, which is used to describe Web Services in general. Both of them, however, do not support simple links and they appear to be too heavy to describe Web APIs. hREST and SA-REST are more approachable as they use microformats which are embedded in the Web page of the API documentations. Although these two approaches are more useful for the semantic correlation, they are not focusing enough on the description itself. RDFa follows the same consideration made for hREST and SA-REST about the specialisation in doing semantic-annotation, turning out to be even more complex to use [19].

For the purpose of this paper we place great emphasis on the analysis of previously listed approaches specific to Web APIs. MicroWSMO relies on hREST offering service property descriptors, but it also focuses on the semantic part of the descriptions. RESTdesc is a logic-based Web API description method that captures the functionality of Web APIs, describing an HTTP request and its preconditions and postconditions expressed in Notation3 (N3), which is a serialization form for the main Semantic Web language, RDF [22]. However, RESTDesc requires manual effort to produce the desired specifications [19] and also there are some complex use cases that cannot be covered, such as cases in smart environments where RDF or N3 are not providing proper solutions. SEREDASj provides a way to describe Web APIs with JSON-based method that is simpler to apply. However, it produces two different documents in order to provide a complete descriptions, proving to be difficult to maintain.

Although there exist API repositories like Programmable Web where users can search for APIs, well-structured API documentations enabling effective discovery are missing. Several frameworks have been introduced to create descriptions for REST APIs through *user-friendly* and *easy-to-use* description format editors [7]. Some REST metadata formats have been created to document REST APIs in a consistent way. These standards offer a way to represent an API by

specifying entry point(s), resource paths, methods to access these resources, parameters to be supplied with these methods, formats of inbound/outbound representations, status codes, error messages and documentary information. Some of the most popular standards are the following: Swagger or Open API Initiative specification, which offers a large ecosystem of API tooling, has a very large community of active users and great support in almost every modern programming languages and allows developers to test the APIs immediately through easy deployment of server instances. API Blueprints, where an API description can be used in the Apiary[10] platform to create automated mock servers, validators etc. The Hydra specification, which is currently under heavy development, tries to enrich current web APIs with tools and techniques from the semantic web area. RAML is a well-structured and modern API modelling language. Swagger is obviously the dominant choice at the moment, though all specifications are promising [17]. We agree with this statement and choose to use Swagger instead of other formats because of its above mentioned promising features and also because it has the capability to provide human-readable API descriptions by using YAML, as well as JSON. Moreover, it defines a standard in the Web API description method, being partner of OAI specification as opposed to other specifications. We agree upon the objective of Open API Initiative, creating an open description format for API services that is vendor-neutral, portable and open to accelerating the vision of a truly connected world. Although API Description created by Swagger editor gives the opportunity to create descriptions easily, they lack detailed information qualifying the properties of an API (e.g. classification of input arguments and response data), which is relevant to address automatic discovery and composition performed by machines. To address these issues, we want to add additional information to the descriptions.

Extensive research has been conducted with the vision to create automatic integration of Web Services or APIs [11]. But in practice most of these approaches are having problems in communicating between candidate Web services or APIs due to the lack of semantic correlation of properties. To automate the interactions between Web APIs there is a need to describe the exchanged data with semantics. To achieve this there are two possibilities, one is by directly creating Web service descriptions following specifications defined in a logic based language, like the Web Ontology Language (OWL) and the second one is by linking existing descriptions to these ontologies (i.e. aligned descriptions to shared domain ontologies). As the first approach needs expertise in logic based languages, its adoption is curtailed. The latter is more approachable because it enriches the existing descriptions to be remain compliant with other semantic descriptions. Thus, this approach reduces the possibilities to lock out non-semantic descriptions.

To support automatic composition of Web Services, several approaches have been proposed focusing on semantic annotations, but many of them are either not validated or the validation lacks credibility [16]. Moreover, most of the existing tools are considering Web Services while we are focusing on Web APIs. For the

[10] https://apiary.io/.

purpose of this paper, we analyse two tools SWEET (Semantic Web sErvices Editing Tool)[11] and Karma[12], which emphasise on user-driven integration of Web APIs by enabling semantic annotations. SWEET is a tool that allows the development of mashup based on linked open data and services, by enabling the creation of semantic descriptions of Web APIs. The input is the HTML Web page describing a Web API and the result is a semantically annotated HTML page, or a RDF MicroWSMO description. Although SWEET allows the definition of semi-automatic annotations, the users have to make long effort because they need to find all the parameters to be annotated in Web APIs description pages. Karma [2, 15] is another tool which allows users to integrate data from different data sources, including databases, spreadsheets, XML, JSON and Web APIs. The inputs to the process are an ontology, a data source and a database of semantic type that the system has learned to recognise, based on prior use of the tool. The system is based on a probabilistic model that is also capable of learning, with a model named conditional random field (CRF) [3], whenever users define a new mapping from data source in the ontology.

Both of these tools guide users in the process of composition of Web APIs and endeavour to suggest the correct annotations, based on the use of ontologies. The main difference between SWEET and Karma is that SWEET allows the addition of hREST tags in the HTML page, since it uses only HTML pages as inputs. Karma, instead, employs a table annotation technique creating a table where, once properties are input into the header row, API responses are populated in the columns. These properties are collected dynamically through different invocations of an API, by defining several different parameters to retrieve the most accurate representation. However, this tool does not consider the context outside tables. We therefore propose to use a different technique: TableMiner [24]. TableMiner is an innovative approach to classify table columns and disambiguate cell contents following different algorithms. This approach considers two types of contexts, one is defined as "in-table context", including column header, column content and row content, and another one is "out-table context", which could include semantic mark-up already inserted in a web page, the web-page title, paragraphs and table captions. The usage of this out-table context and the previously mentioned algorithms are taking TableMiner a step forward in the State-of-the-Art:

- first, it adopts a boostrapping, incremental approach to interpret columns with at least 51 % of non-empty rows and with mostly named entities;
- then, a forward-learning process uses an incremental inference with a stopping algorithm that makes a first semantic association with the contents of columns, followed by a process of disambiguation of the contents in the cells and the searching of the highest scoring entities which could represent the right concepts;
- at this point, a backward-update step kicks in to make an interpretation of the remaining data, guided by previously obtained results. This phase could

[11] http://sweet.kmi.open.ac.uk/index.html.
[12] http://usc-isi-i2.github.io/karma/.

modify the columns classification since there are new disambiguated entity content cells;
- finally, classifications and disambiguated entities are updated again with a mutually recursive pattern until they can be considered stabilised.

Another strong point in favour of TableMiner is the usage of predefined incremental inference with stopping algorithm, which does not require to analyse all the rows of a column, instead it stops when it feels confident, reducing considerably the computation time. Finally, TableMiner is adaptable to any knowledge bases.

```
1. info:
2.   title: Bike sharing API
3. host: api.bikesharing.com
4. produces:
5.   - application/json
6. paths:
7.   /bikes:
8.     get:
9.       description: |
10.        The Bikes endpoint returns information about
   available bikes at the nearest position.
11.      parameters:
12.        - name: lat
13.          in: query
14.          description: Latitude component of location.
15.          required: true
16.          type: number
17.          format: double
18.        - name: lng
19.          in: query
20.          description: Longitude component of location.
21.          required: true
22.          type: number
23.          format: double
24.      tags:
25.        - Bikes
26.      responses:
27.        200:
28.          description: An array of bikes
29.          schema:
30.            type: array
31.            items:
32.              $ref: '#/definitions/Bike'
```

```
1. info:
2.   title: Weather API
3. host: api.weather.com
4. produces:
5.   - application/json
6. paths:
7.   /weather:
8.     get:
9.     tags:
10.        - weather
11.      responses:
12.        200:
13.          description: An array of geographical area with
   weather indication
14.          schema:
15.            type: array
16.            items:
17.              $ref: '#/definitions/GeoArea'
```

Fig. 1. The APIs descriptions created following OAI specification

3 API Descriptions

To analyse how we can enrich the existing REST description formats, let's discuss OAI descriptions that follow the Swagger format by providing an example involving *end-user developers*. Assume that a couple of tourists, John and Mary just arrived at one of the airports near Milan to visit the city. To move around they have different alternatives: (i) public transports, or sharing services for (ii) cars or (iii) bikes. Mary and John want to choose one of them according to preferences and/or context (e.g. weather, time, location, accessibility, etc.). Unfortunately, they have to invoke different information services to collect data before making an informed decision. Moreover, data are often not easily comparable or complete: for example, in the descriptions of bike sharing and weather APIs (Fig. 1) spatial references are in different formats and with different meaning (e.g., longitude/latitude versus area by points). Furthermore, most of the

API descriptions are available only as HTML web pages, yet this is not adequate to support (semi) automatic comparison and composition of properties. We propose to extend OAI descriptions by mapping properties using DCAP to deliver API profiles, and defining a method for automatic extraction of concepts from shared vocabularies.

As for REST APIs, we should consider the use of HTTP OPTIONS method[13] to make REST APIs self-descriptive. Currently, the OPTIONS method is basically used to retrieve simple information about a resource, like the available HTTP methods that can be used in the communication with the specific API; however, since there is no standard response to an OPTIONS request, a full description could be returned in the response-body. With our approach API descriptions can be edited using the tool we developed, as explained in Sect. 5, and retrieved by invoking OPTIONS method. Other approaches that make the most of HTTP OPTIONS method are [13,21].

3.1 API Profiles

As defined in RFC6906[14] "a profile is not to alter the semantics of the resource representation itself, but to allow clients to learn about additional semantics (constraints, conventions, extensions) that are associated with the resource representation, in addition to those defined by the media type and possibly other mechanisms" [23]. Given this definition it can be stated that API profile documents can offer a view of what is supplied by an API, and how clients and servers can expose features in a machine-readable format. The Dublin Core Metadata Initiative[15] (DCMI) released the DCAP format to describe profile metadata defining the constraints on how the RDF vocabularies are used to create profiles by linking properties. DCAP has been developed following the concept of Metadata Profile[16] that supports additional descriptive information about the contents of the response (e.g. useful indexing properties of the document, terms of use, etc.). For example in Fig. 2, line 8 and 12 link spatial data to concepts of latitude (lat) and longitude (long) from shared vocabularies. The approach is to enrich existing descriptions (Fig. 1) with such explicit references to shared vocabularies (Fig. 2) to facilitate automatic composition.

4 Semantic Annotations in API Descriptions

As standard description formats are missing, we propose to add semantic annotations in API descriptions through API profiles by linking properties to concepts in shared vocabularies. To show how the proposed approach works, let's go back to our example: Mary and John may save time and effort if we can provide all information related to useful services (e.g. public transport in Milan) in an

[13] https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.
[14] https://www.ietf.org/rfc/rfc6906.txt.
[15] http://dublincore.org/.
[16] https://www.w3.org/TR/html4/struct/global.html.

```
1. Description template: Bike id=bike
2.   minimum = 0; maximum = unlimited
3.   Statement template: model
4.     Property: http://dbpedia.org/property/name
5.     minimum = 0; maximum = 1
6.     Type of Value = “literal”
7.   Statement template: lat
8.     Property: http://www.w3.org/2003/01/geo/wgs84_pos#lat
9.     minimum = 1; maximum = 1
10.    Type of Value = “float”
11.  Statement template: lng
12.    Property: http://www.w3.org/2003/01/geo/wgs84_pos#long
13.    minimum = 1; maximum = 1
14.    Type of Value = “float”
15.    [..]
```

Fig. 2. An example DCAP profile

```
geo: http://www.w3.org/2003/01/geo/wgs84_pos#

1. definitions:                        OAI specification
2.   Bike:
3.     type: object                    DCAP profile
4.     properties:
5.       bike_id:
6.         type: number
7.       description:
8.         type: string
9.       lat:
10.        type: number
11.        property: geo:lat
12.        type_of_value: float
13.      lng:
14.        type: number
15.        property: geo:lng
16.        type_of_value: float
```

Fig. 3. Adding API profile using DCAP specification in OAI specification

aggregated way by composing descriptions. For example, if they can compare weather forecast information with available mobility services, they can make informed decisions like using bike sharing service in case of a sunny day, or a car sharing service in case of rain. To improve the descriptions in Fig. 1, we propose to add API profile information by means of DCAP specifications, as shown in Fig. 3 that specifies a definition of a bike. The resulting description consists of a comprehensive set of information that facilitate composition of responses. For example, in order to identify the best solution for Mary and John, car and bike sharing API responses can be enriched with contextual information such as weather forecast, traffic congestion, accessibility for disabled or elders, etc. In Fig. 4, the position of a bike is uniquely identified by concepts *geo:lat* and *geo:lng*. Similarly, information about the weather have been linked to concept *schema.org/geo*. In this way, it is possible to compare the responses of the two APIs and find the weather conditions in the area in which the bike is located. Similarly, when the visitors want to use a car, they need to know if a parking place is available near their destination. They may use Google Places API by giving the value *parking* in the *types filter* for parking place searches. This API

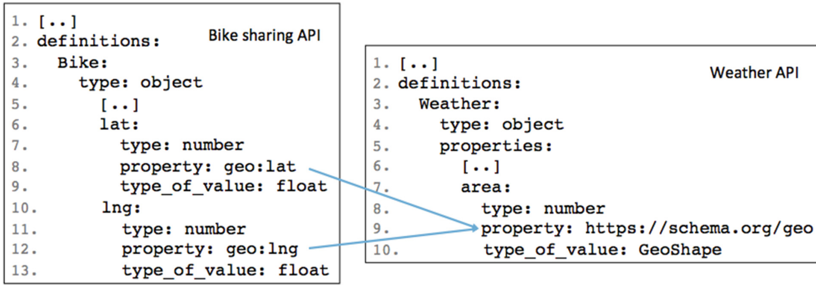

Fig. 4. Mapping of properties between responses of Bike sharing and Weather APIs

provides responses with the *address* property that shows geocoding results with a precise position. By correlating the semantics of properties of Weather API, *schema.org/geo*, and Google Places API, *schema.org/address*, the visitors can get required information to take a decision such as whether to book a open or closed parking place.

From the discussion in Sect. 2, we find an issue that developers often have to manually define the mappings between the information consumed and produced by Web APIs to shared vocabularies. In the following section we define the architecture of the tool that can address the discussed issues.

5 The Architecture - How the System Works

We have developed a system that target both *professional developers*, who have technical expertise in developing applications, and *end-user developers* that may not be familiar with the technologies discussed in the previous sections. Therefore, the aim is to create an abstraction layer to hide the technological complexity to the end-users and make the task of composing descriptions and services easier. The resulting system makes available a composition process through a REST API, which is provided by the *API Provider* component (Fig. 5).

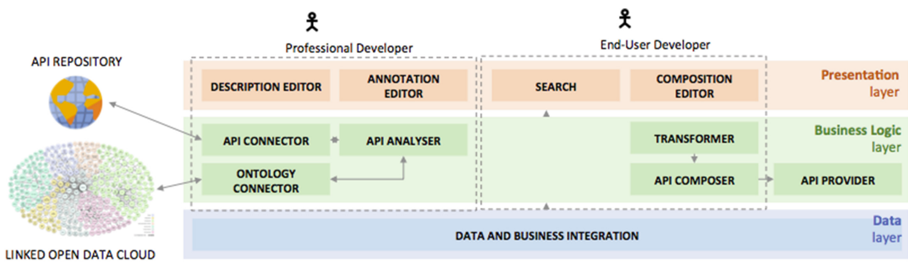

Fig. 5. The system architecture

The system architecture (Fig. 5) consists of three layers: *Presentation layer*, *Business logic layer* and *Data layer*. Both Presentation and Business Logic layers consist of components dedicated to different group of users: *Professional Developers* and *End-User Developers*.

The presentation layer dedicated to *professional developers* includes components that provide a user-friendly interface to enrich the descriptions of Web APIs, and manage semantic annotations. These tasks have been accomplished by extending the Swagger Editor, both for descriptions and annotations. In particular, the description process is semi-automatically managed by augmenting existing API descriptions, which can be retrieved from existing repositories (e.g. ProgrammableWeb), services registry, or by exploiting the HTTP OPTIONS method, as discussed in Sect. 3. Note that the OPTIONS method is also exploited by the system to support maintenance and evolution of descriptions already known by the system. If it is not possible to retrieve any initial description, the developer can insert a new API description manually using the Description Editor. These descriptions are represented in JSON or YAML format. For each resource, all relevant information such as available HTTP operations (e.g. GET, POST, PUT, DELETE, OPTIONS, HEAD, PATCH), the list of parameters for each operation, possible responses are collected. The process of creating a description is detailed in Algorithm 1.

The Business Logic Layer allows developers to semi-automatically add semantic annotations to inputs and outputs by following the approach discussed in [24,25]. The system automatically builds a table by putting properties in the header row, and filling up columns with API responses. These responses are collected dynamically by the *API Connector* component through multiple invocations of the involved APIs. The use of different input values allows the building of an accurate description of the APIs. In case of failure, the developer is asked to provide valid inputs to proceed. For the Bike sharing API example presented in Sects. 3, and 4, the header rows include input (e.g., "id_station") and output (e.g., "station name", "lat", "long", "free bikes", "total slots") properties, and the cell contents will be incrementally filled with data of each API invocation. So, the system is able to break up the response code (e.g. JSON, XML) in order to identify the output properties and their values.

Algorithm 1. Retrieve or create API description

Result: API description

1 **if** *description is available* **then**
2   retrieve description from existing repositories, services registries or via OPTIONS method;
3 **else**
4   create it manually using the Description Editor;

Algorithm 2. Create and add API profiles to API descriptions

```
   Data: API description
   Result: API description with API profile
 1 Detect all resources' end-point;
 2 foreach end-point do
       // collect data
 3     repeat
 4         generate input parameters following the API description;
 5         if input parameters cannot be generated then
 6             take input parameters from the user
 7         invoke API with input parameters;
 8         collect results;
 9     until at least N results are collected        /* default N=10 */;
       // create tables
10     foreach results do
11         create a header row with API properties;
12         fill content-cells with values from inputs and responses;
   // add semantic annotations
13 foreach tables do
14     apply TableMiner technique;
15     show table to the user;
16     if table annotation is not complete then
17         show related vocabularies and/or alternatives to the user;
18         ask the user to manually add links;
19     if the user wants to review the annotations then
20         show related vocabularies to the user;
21         let the user confirm or modify the links;
22     insert API profile in API description;
```

In the Mary and John example, *lat*, *lng* and *area* are in the header row; values like "45.523", "9.219" and "[[45.524902, 9.216672], [45.526398, 9.218571], ...]", which is an array of spatial points, fill up the column cells.

Algorithm 2 defines the process, which extends the one proposed by Karma [14] with the use of TableMiner technique to analyse the semantic properties of the resulting table. The TableMiner annotation technique is applied by the *API Analyzer* component that follows the steps described in Sect. 2 to set the meaning of properties by automatically linking them to concepts in the shared vocabularies. However, if the semi-automatic process fails, the developer is able to provide semantic annotation manually using the editor. Moreover, the system can support the developers by showing shared domain vocabularies and annotation alternatives.

Finally, the *Data and Business Integration* component stores the data to support the analysis of the APIs, and the produced descriptions and semantic

annotations, to support the creation of an *ecosystem* of services. This *ecosystem* is an open set of services that could be automatically retrieved and linked each others to be able to follow the evolving user needs. The descriptions are retrievable by the OPTIONS method, as already discussed, to support use and evolution. In the context of this paper, we consider Mary and John as information seekers who want to know some information to facilitate their mobility. They are representative of generic users who wants to know more information about specific services (e.g., mobility) and related services. They are provided with descriptions to select and compose APIs according to patterns that have been pre-defined by professional developers.

The *Composition Editor* component is devoted to create compositions of services. The first step is to search for Web APIs that are already stored in the system by using the *Search* component. All relevant results and their possible combinations are loaded and showed in the interface to let the user select the APIs that match a set of given requirements. The second step is to create compositions either by directly linking the outputs and inputs of selected APIs, or by including *transformation* services that transform and make outputs compatible to inputs according to the semantic relations hold in the annotations. The *Transformer* component has the task of managing the set of transformation rules that make properties compatible. Composition patterns are then stored and provided to users such as Mary and John that have the task to populate such patterns with the services of interests.

The Mary and John example can provide a general understanding of how these transformation rules work: after *lat* becomes *geo:lat*, and *lng* becomes *geo:lng* to build the augmented description (shown in Fig. 4), it is possible to identify which area includes the given pair of *geo:lat* and *geo:lng* values by applying the transformation rules. In such a way, it is possible to identify the weather conditions in the area in which a bike station is currently located. If the system cannot identify the appropriate rules to manage some annotated properties, the developer can insert new ones to enrich the system and ensure its evolution. In other words, the system provides end-users with synthetic information to accomplish a given task, anyway, if they are interested to see more details they can explore the process of transformation and composition. Moreover, if the user has the needed skills, he or she can contribute to the system evolution by adding transformation rules and/or composition patterns. One of the major advantage of the proposed system remains the separation of the annotation activities, which requires skills on semantic technologies, from the transformation and composition activities, which requires basic programming skills, or even no particular skills to just use the system as it is, like in the case of Mary and John.

6 Conclusion

Today, Web APIs are associated with textual descriptions that are not understandable by machines and cannot be composed (semi) automatically. There exist approaches, including WADL, WSMO Lite, Resource-Oriented Service

Model (ROSM) and RESTdesc, that provide rich semantic descriptions, but they are not widely adopted because of the required expertise in Semantic Web Languages (e.g. RDF, SPARQL, N3) as well as in-depth domain knowledge [19]. Although machine-readable descriptions (e.g. MicroWSMO, Minimal Service Model, SA-REST) have been introduced to support additional semantic information, tools for creating automatic or semi-automatic semantic annotations are missing. Such shortcomings motivate our work and our long term goal of defining an abstract layer on top of API descriptions and profiles to hide the intrinsic complexity to the end-users.

The current contribution is an extension to Open API Initiative (OAI) Specification following the Dublin Core Application Profile (DCAP) guidelines. Target users are technology experts and professional developers that can understand the involved concepts and drive the semi-automatic tool we developed. The next step is to introduce high-level concepts that target specific requirements (e.g. common needs and requirements of specific groups of users), and are understandable by generic users. Such concepts will be implemented in visual interfaces that can support actual user evaluation tests.

References

1. Danielsen, P.J., Jeffrey, A.: Validation and interactivity of web API documentation. In: 2013 IEEE 20th International Conference on Web Services (ICWS), pp. 523–530. IEEE (2013)
2. Gupta, S., Szekely, P., Knoblock, C.A., Goel, A., Taheriyan, M., Muslea, M.: Karma: a system for mapping structured sources into the semantic web. In: Simperl, E., Norton, B., Mladenic, D., Valle, E.D., Fundulaki, I., Passant, A., Troncy, R. (eds.) ESWC 2012. LNCS, vol. 7540, pp. 430–434. Springer, Heidelberg (2012)
3. Lafferty, J., McCallum, A., Pereira, F.C.: Conditional random fields: probabilistic models for segmenting and labeling sequence data, pp. 282–289 (2001)
4. Lanthaler, M., Gütl, C.: A semantic description language for restful data services to combat semaphobia. In: Digital Ecosystems and Technologies Conference (DEST), 2011 Proceedings of the 5th IEEE International Conference on Digital Ecosystems and Technologies, pp. 47–53. IEEE (2011)
5. Lanthaler, M., Gütl, C.: Hydra: a vocabulary for hypermedia-driven web APIs. In: LDOW 996 (2013)
6. Mayer, S., Inhelder, N., Verborgh, R., Van de Walle, R., Mattern, F.: Configuration of smart environments made simple: combining visual modeling with semantic metadata and reasoning. In: Internet of Things (IOT), 2014 International Conference on the Internet of Things, pp. 61–66. IEEE (2014)
7. Mitra, R.: Rapido: a sketching tool for web API designers. In: Proceedings of the 24th International Conference on World Wide Web Companion, pp. 1509–1514. International World Wide Web Conferences Steering Committee (2015)
8. Myers, B.A., Jeong, S.Y., Xie, Y., Beaton, J., Stylos, J., Ehret, R., Karstens, J., Efeoglu, A., Busse, D.K.: Studying the documentation of an API for enterprise service-oriented architecture. IGI Global (2012)

9. Panziera, L., De Paoli, F.: A framework for self-descriptive restful services. In: Proceedings of the 22nd International Conference on World Wide Web Companion, pp. 1407–1414. International World Wide Web Conferences Steering Committee (2013)
10. Schmachtenberg, M., Bizer, C., Paulheim, H.: Adoption of the linked data best practices in different topical domains. In: Mika, P., et al. (eds.) ISWC 2014, Part I. LNCS, vol. 8796, pp. 245–260. Springer, Heidelberg (2014)
11. Sheng, Q.Z., Qiao, X., Vasilakos, A.V., Szabo, C., Bourne, S., Xu, X.: Web services composition: a decades overview. Inf. Sci. **280**, 218–238 (2014)
12. Sheth, A.P., Gomadam, K., Lathem, J.: SA-REST: semantically interoperable and easier-to-use services and mashups. IEEE Internet Comput. **11**(6), 91 (2007)
13. Steiner, T., Algermissen, J.: Fulfilling the hypermedia constraint via HTTP OPTIONS, the HTTP vocabulary in RDF, and link headers. In: Proceedings of the Second International Workshop on RESTful Design, pp. 11–14. ACM (2011)
14. Taheriyan, M., Knoblock, C.A., Szekely, P., Ambite, J.L.: Rapidly integrating services into the linked data cloud. In: Cudré-Mauroux, P., et al. (eds.) ISWC 2012, Part I. LNCS, vol. 7649, pp. 559–574. Springer, Heidelberg (2012)
15. Taheriyan, M., Knoblock, C.A., Szekely, P., Ambite, J.L.: Semi-automatically modeling web APIs to create linked APIs. In: Proceedings of the ESWC 2012 Workshop on Linked APIs (2012)
16. Tosi, D., Morasca, S.: Supporting the semi-automatic semantic annotation of web services: a systematic literature review. Inf. Softw. Technol. **61**, 16–32 (2015)
17. Tsouroplis, R., Petychakis, M., Alvertis, I., Biliri, E., Lampathaki, F., Askounis, D.: Community-based API builder to manage APIs and their connections with cloud-based services. In: CAiSE Forum (2015)
18. Vandenbussche, P.Y., Atemezing, G.A., Poveda-Villalón, M., Vatant, B.: Linked open vocabularies (LOV): a gateway to reusable semantic vocabularies on the web. Semant. Web (Preprint) 1–16 (2015)
19. Verborgh, R., Harth, A., Maleshkova, M., Stadtmüller, S., Steiner, T., Taheriyan, M., Van de Walle, R.: Survey of semantic description of REST APIs. In: Pautasso, C., Wilde, E., Alarcon, R. (eds.) REST: Advanced Research Topics and Practical Applications, pp. 69–89. Springer, New York (2014)
20. Verborgh, R., Mannnens, E., Van de Walle, R.: Bottom-up web APIs with self-descriptive responses. In: Proceedings of the First Karlsruhe Service Summit Workshop-Advances in Service Research, p. 143. KIT Scientific Publishing (2015)
21. Verborgh, R., Steiner, T., Van Deursen, D., De Roo, J., Van de Walle, R., Vallés, J.G.: Description and interaction of restful services for automatic discovery and execution. In: 2011 FTRA International Workshop on Advanced Future Multimedia Services (AFMS 2011). FTRA (2011)
22. Verborgh, R., Steiner, T., Van Deursen, D., De Roo, J., Van de Walle, R., Vallés, J.G.: Capturing the functionality of web services with functional descriptions. Multimedia Tools Appl. **64**(2), 365–387 (2013)
23. Wilde, E.: The "profile" link relation type. https://www.ietf.org/rfc/rfc6906.txt. Accessed 24 May 2016
24. Zhang, Z.: Start small, build complete: effective and efficient semantic table interpretation using tableminer. Under Transpar. Rev.: Semant. Web J. (2014)
25. Zhang, Z.: Towards efficient and effective semantic table interpretation. In: Mika, P., et al. (eds.) ISWC 2014, Part I. LNCS, vol. 8796, pp. 487–502. Springer, Heidelberg (2014)