A Practical Approach to Services Composition Through Light Semantic Descriptions

Chapter *in* Lecture Notes in Computer Science · August 2018

DOI: 10.1007/978-3-319-99819-0_10

2 authors:

Marco Cremaschi
Università of Milano - Bicocca
38 PUBLICATIONS **206** CITATIONS
SEE PROFILE

Flavio De Paoli
Università degli Studi di Milano-Bicocca
146 PUBLICATIONS **1,249** CITATIONS
SEE PROFILE

A Practical Approach to Services Composition Through Light Semantic Descriptions

Cremaschi M., De Paoli F.

Cremaschi M., De Paoli F. (2018) A Practical Approach to Services Composition Through Light Semantic Descriptions. In: Kritikos K., Plebani P., de Paoli F. (eds) Service-Oriented and Cloud Computing. ESOCC 2018. Lecture Notes in Computer Science, vol 11116. Springer, Cham

A practical approach to services composition through light semantic descriptions

Marco Cremaschi and Flavio De Paoli

Department of Informatics, Systems and Communication
University of Milan - Bicocca, Viale Sarca 336/14, Milan, Italy
{cremaschi,depaoli}@disco.unimib.it

Abstract. Services composition has been much investigated over the last decade without reaching shared and consolidated results mainly for the lack of interoperable descriptions of services and the consequent need of extensive user intervention. In this paper, we propose a light and practical approach to create machine-readable descriptions of output data that can be merged or used (as-is or adapted) as input data to other services. The solution relies on the popular and standard OpenAPI descriptions augmented with annotations based on JSON-LD format. Services descriptions are created by table annotations techniques applied on sets of given or retrieved output values. The approach has been implemented in a tool and validated with a set of real services.

1 Introduction

In the last decade, we have witnessed the evolution of web services models from the WSDL/SOAP to the REST. This change is tangibly visible, for example, by searching ProgrammableWeb[1], perhaps the largest repository of web descriptions. One of the reasons for this evolution is the need to simplify the service reference model to enhance comprehensibility and standardisation, and therefore provide the bases for automatic management of descriptions and composition. A similar evolution is needed in the realm of semantic web services. As a matter of facts, well-defined proposals that deliver machine-readable descriptions, such as OWL-S: Semantic Markup for Web Services [10], Semantic Annotation for WSDL and XML Schema (SA-WSDL) [7], Micro Web Service Model Ontology (MicroWSMO) [6] and Semantic Annotations for REST (SA-REST) [5], failed to become widely used mainly for their complexity that requires the involvement of experts.

Current description models address services accessible through API REST, and provide meta-languages to describe services as documents based on *property-*

The work presented in this paper has been partially supported by the EU H2020 project EW-Shopp - Supporting Event and Weather-based Data Analytics and Marketing along the Shopper Journey - Grant n. 732590.

[1] https://www.programmableweb.com/apis/directory

value pairs. OpenAPI Specification[2], also known as Swagger[3], API Blueprint[4] and RAML[5] are the most representative. However, these models do not support semantic annotations to make *property-value* pairs interoperable. In this paper, we discuss an extension of the popular OpenAPI model to add semantic annotations on input parameters and output properties of services. Such annotations are compliant with the JSON-LD[6] format to follow the REST philosophy in order to minimise the user involvement in many practical situations.

The availability of semantic descriptions of APIs enables the development of automatic techniques and tools to support services composition [13]. A general definition states that a process of composition is defined as the aggregation of different Web services into a single compound service to perform more complex functions [14]. In this context, we refer to information services and the mash-up of *results* got from independent services to deliver comprehensive answers to users' requests, or to prepare data coming from a set of services to invoke another service. We call the former *merge composition* and the latter *sequence composition*. Merge composition involves more services that are invoked in parallel with the same input data, whose answers are then composed. Sequence composition involves a service which is invoked with input data coming from the composition of answers from one (adaptation) or more (mash-up) services. This work roots and extends the one presented in [3, 8] by proposing a formalised model to create semantic descriptions for Web APIs, and a set of composition rules based on semantic annotations inside the descriptions. Moreover, we implemented the AutomAPIc tool to support users in the creation and composition of semantic descriptions.

Services composition may occur at design time or at runtime. At design time, the ability of automatic processing of descriptions enables actors (users or machines) to discover, select and compose services. If semantic descriptions are not available, actors can rely on techniques, such as table interpretation and NLP techniques, to build such missing descriptions. At runtime, composition supports adaptation and substitution of services to ensure contextualization and accomplishment of tasks.

In the next section, we discuss services description and composition to motivate the work. Then, Section 3 describes the proposed extension of the OpenAPI model to include semantic annotations. Section 4 discusses the composition techniques in the split and sequence cases. Section 5 presents the tool that provides full support to users to manage the process of building descriptions and composing services. Section 6 validates the approach by addressing a set of real services. Finally, Section 7 draws some conclusions.

[2] https://www.openapis.org
[3] http://swagger.io
[4] https://apiblueprint.org
[5] http://raml.org
[6] https://json-ld.org

2 Services description and composition

In the last decade, the composition of services has been widely investigated without getting to effective results for many reasons. Among others, the most relevant are the use of different architectural styles, the unexpected evolution of services, and the use of different description languages and different conceptual models [12]. Moreover, composition may occur at the design stage, leading to *static* compositions, or at runtime, leading to *dynamic* composition. The latter is best suited to address the issues in real environments that change continuously and requires automatic tools to search for, select and compose Web services automatically. The main issue affecting automatic composition is the limited number of available machine-readable descriptions associated with services.

A traditional way to compose services is the use of orchestration languages, such as BPEL (Business Process Execution Language) [16] or OWL-S (Ontology Web Language for Services) [10], which support the manual definition of abstract processes that can be implemented by actual services. On the other side, dynamic composition in *automatic* way can be achieved by exploiting the semantic Web and the planning techniques. However, the realisation of a completely automatic composition process is complex and presents several issues [14]. The main problems are the missing of semantics associated with services, and the capability of understanding the semantics even when present.

The most popular syntactic description model is WSDL 2.0 (Web Services Description Language) [1], which defines an XML format for describing Web services by separating the abstract functionality offered by a service from concrete details such as how and where that functionality is offered. Although it supports descriptions of both SOAP-based services, and REST/API services, it is the de-facto standard for the former but is rarely adopted for the latter. The Web Application Description Language (WADL) is a machine-readable XML format that was explicitly proposed for API services. WADL was also proposed for standardisation, but there was no follow-up.

Recently, *user-friendly* and *easy-to-use* metadata formats have been introduced, along with editors to support developers in the creation of descriptions for REST APIs. Among others, popular description formats are the Open API Specification, which provides human-readable API descriptions based on YAML and JSON. RAML is a YAML-based language for describing RESTful APIs. API Blueprint is a documentation-oriented web API description language, which provides a set of semantic assumptions laid on top of the Markdown syntax. The Hydra specification, which is currently under massive development, aims to enrich current web APIs with tools and techniques from the semantic web area.

Table 1 is an extension of the one presented in [15] to compare the number of questions posed in Stack Overflow and the number of Git stars (showing appreciation to a project) received by the four description models under study. The increasing number of available descriptions highlights the growing popularity of descriptions, and the relevance of tools that support the creation, publication, use and maintenance of service descriptions. The common limitation of such models is the lack of semantic descriptions, which motivated our previous paper

[3]. In order to be effective, we extended the most popular model, OpenAPI, to support semantic-enabled tools for describing, discovering, and then compose APIs.

Table 1. Comparison of API description models (at May 27, 2018).

Detail/Model		*API Blueprint*	*RAML*	*WADL*	*OpenAPI Spec*
Format		Markdown	YAML	XML	YAML, JSON
Licence		MIT	ASL2.0	Sun	ASL 2.0
Version		Format 1A revision 9	1.0.1	31 August 2009	3.0.1
Initial commit		Apr 2013	Sep 2013	Nov 2006	Jul 2011
Pricing plan		Yes	Yes	No	No
StackOverflow Questions	2015	88	153	86	13
	2016	61	168	84	166
	2017	40	174	74	319
	2018	15	56	33	218
Github Stars	2015	1,819	1,058	N/A	2,459
	2016	X	X		X
	2017	5,390	2,735		6,360
	2018	6566	3060		9836
Google Search		985K	1M	486K	8M

3 A light semantic Web API description model

The OpenAPI is the most promising description model since it defines a simple format to specify descriptions supported by a broad set of vendor-neutral API tools, whose development involves a massive community of active users. Such tools provide significant support to almost every modern programming languages to create and test APIs. Moreover, the OpenAPI Initiative is an open source project sustained by relevant stakeholders, including Google, IBM, Microsoft and PayPal. There are several repositories collecting API REST described using OpenAPI, such as *SmartAPI*[7] and *APIs.guru*[8].

An OpenAPI description is a YAML or JSON document that contains a list of resources and a list of operations that can be applied to those resources. An example is provided in Listing 1.1, which describes the Google Books API. Notice that the API is described by *name:value* pairs of strings without any semantics.

We propose to extend such descriptions by inserting annotations (i.e., links to ontology classes and ontology properties) through the use of the JSON-LD[9] format. JSON-LD provides (i) a universal identification mechanism for JSON objects through the use of Internationalized Resource Identifiers (IRIs); (ii) a way to disambiguate shared keys between different JSON documents through IRIs mapping and context; (iii) the possibility to annotate the strings with indications on the used language; and (iv) a way to associate data types with values (e.g., dates, times, etc.).

[7] http://smart-api.info/registry
[8] http: https://apis.guru/openapi-directory/
[9] https://json-ld.org/spec/latest/json-ld/#basic-concepts

Listing 1.1. OpenAPI description of the Google Books API.

```
"paths": {
"/volumes": {
    "get": {
        "parameters": [{
            "name": "title", [...]
        }],
       },
        "responses": {
            "200": {
                "schema": {
                    "title": "result",
                    "type": "object",
                    "properties": {
                        "isbn": { "type": "string"},
                        "author": { "type": "string" },
                        "title": { "type": "string" }, [...]
```

The marriage between JSON-LD and OpenAPI descriptions occurs through the introduction of the *semanticAnnotations* property (e.g., Listing 1.2, line 8 and 27), which is composed of two parts: the definition of a context, by the keyword *@context* (e.g., line 9 and 28), to set short names for the reference ontologies used throughout the description; and a list of annotations for parameters (input values) and responses (output values). Each annotation is a pair to annotate the *name*, introduced by the keyword *@id* (e.g., line 14 and 33), and the *value*, introduced by the keyword *@type* (e.g., line 15 and 34). Annotations are IRIs that uniquely identify elements.

Listing 1.2. Semantic OpenAPI description of the Google Books API.

```
"basePath": "/books/v1",
"paths": {
"/volumes": {
    "get": {
        "parameters": [{
            "name": "title", [...]
        }],
        "semanticAnnotations": {  /** Input semantics **/
            "@context": {{
                "dbp": "http://dbpedia.org/property/",
                "xsd": "http://www.w3.org/2001/XMLSchema#"
            },
            "title": {
                "@id": "dbp:title",
                "@type": "xsd:string"
            }
        },
        "responses": {
            "200": {
                "schema": {
                    "type": "object",
                    "properties": {
                        "isbn": { "type": "string" },
                        "author": { "type": "string" },
                        "title": { "type": "string" }
                    },
                    "semanticAnnotations":{  /** Output semantics **/
                        "@context": {
                            "dbp": "http://dbpedia.org/property/",
                            "xsd": "http://www.w3.org/2001/XMLSchema#"
                        },
                        "isbn": {
                            "@id": "dbp:isbn",
                            "@type": "xsd:integer"
                        },
                        "author": {
                            "@id": "dbp:author",
                            "@type": "xsd:string"
                        },
                        "title": {
                            "@id": "dbp:title",
                            "@type": "xsd:string"
                        }, [...]
```

4 Composition types and rules

In this context, we consider the composition of information services and interested in mashing up results from independent services to deliver a comprehensive answer to users' requests, or to prepare data coming from a set of services to invoke another service. We call the former *merge composition* and the latter *sequence composition*. Merge composition involves more services that are invoked in parallel with the same input data, and the results are composed [11]; while sequence composition involves a service which is invoked with input data that are coming from one (data adaptation) or more (data mash-up) services.

Dealing with automatic *sequence composition*, semantic compatibility needs to be verified. In this context, semantic compatibility occurs when a semantic relationship holds between the semantic classes[10] of output properties of an API and input parameters of another API. In such cases, output properties can be used as input parameters, possibly after some transformations (Figure 1).

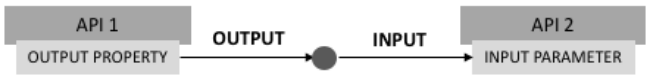

Fig. 1. Schema of sequence composition.

To evaluate semantic compatibility, we can define four rules:

Rule 1: single ontology, same concepts. If annotations refer to the same ontology, and name/value pairs refer to the same concept, or two concepts in relation *owl:sameAs*, then the composition is straightforward since they are compatible (see Figure 2.1).

Rule 2: different ontologies, same concepts. If annotations refer to different ontologies (see Figure 2.2), we need to verify if the annotations of involved name/value pairs are *equivalent* (i.e., they refer to the same ontology concepts or property). For example, some ontologies such as DBPedia[11] and Wikidata[12] provide the properties *owl:equivalentProperty* and *owl:equivalentClass* to address the issue. These properties, however, are not supported by all ontologies, therefore some *Ontology matching* [4] techniques may need to be exploited to check for compatibility.

Rule 3: single ontology, different concepts in relation to each other. If annotations refer to the same ontology, and name/value pairs refer to different ontology concepts or properties, then values' compatibility need to be checked. If between the involved concepts relations such as *subclass* and *subproperty* hold, then they may be compatible and the composition may occur. An example is shown in Figure see Figure 2.3, where the annotation *@type:*

[10] https://www.w3.org/TR/owl2-syntax/#Classes
[11] https://dbpedia.org
[12] https://www.wikidata.org

dbp:zipCode refers to a subproperty of *dbp:postalCode*. Therefore, API 1 and API 2 are compatible.

Rule 4: different concepts not related to each other. If annotations of the name/value pairs refer to different ontology concepts or properties in the same ontology or different ontologies, and among these elements none of the above rules apply, compatibility may occur after a transformation (e.g., by invoking a third-party service). For example (see Figure 2.4), if API 1 returns a mail address, and API 2 requires latitude and longitude values as input parameters, then a third API is needed to perform the conversion.

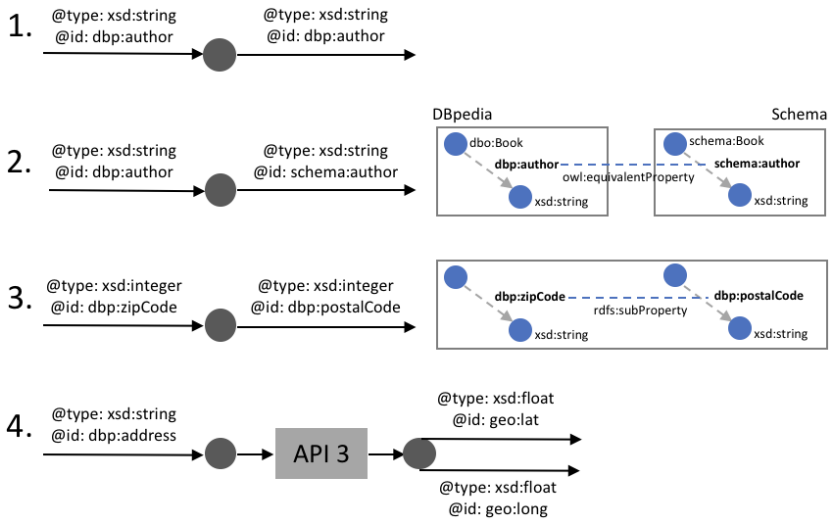

Fig. 2. Sequence composition: examples of the four compatibility cases.

Let's consider a use case to discuss the composition rules described above. Assume we seek an application that helps students to retrieve information to access textbooks. The application should provide information about different options: bookshops or e-commerce purchase, library consultation, or free download. The composition related to this use case is shown in Figure 3: we consider a process that starts with Google Books API, which gets a title in input and delivers a full report about accessing the requested book in output.

A first example of *sequence* composition type, is the service that collects information about a book from Google Books API[13] and calls Amazon Market API[14] to check if it is available. The Semantic OpenAPI Description of Amazon

[13] https://developers.google.com/books/
[14] https://developer.amazonservices.it/gp/mws/docs.html

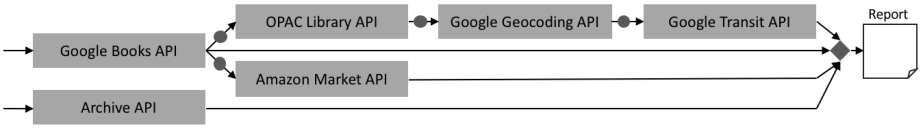

Fig. 3. Example of a process of composition of the use case.

Market API is in Listing 1.3. The semantic annotation in line 6 finds a correspondence in the description of the Google Books API, in line 33 of Listing 1.2; in both descriptions the concept of *ISBN* is described with the same semantic annotation. Therefore, the services can be composed (rule 1).

Listing 1.3. The input part of the description of the Amazon Market API.

```
1 "get": {
2     "parameters": [{
3         "name": "IsbnItem", [...]
4     }],
5     "semanticAnnotations": {
6         "IsbnItem": {
7             "@id": "dbp:isbn",
8             "@type": "xsd:integer"
9         }, [...]
```

A second example is the sequence composition of the Google Books API, the Library API, and the Google Transit API: first the Library API is invoked to check the presence and availability of the book, and then the Google Transit API is invoked to check the existence of public transport to reach the library.

The composition of Google Books API and the Library API can be performed according to rule 1, and rule 3. The annotations on line 8 and line 16 of Listing 1.4 are compatible with the annotations in line 32 and 40 of Listing 1.2 (rule 1). The parameter on line 12 of Listing 1.4 is compatible with the property present in line 36 of Listing 1.2 since the relation *rdfs:SubPropertyOf* holds between them (rule 3).

Listing 1.4. Extract from the description of the Library API.

```
1  "get": {
2      "parameters": [
3          { "name": "Isbn" },
4          { "name": "author" },
5          { "name": "title" }
6      ],
7      "semanticAnnotations": {
8          "Isbn": {
9              "@id": "dbp:isbn",
10             "@type": "xsd:integer"
11         },
12         "author": {
13             "@id": "dbp:writen",
14             "@type": "xsd:string"
15         },
16         "title": {
17             "@id": "dbp:title",
18             "@type": "xsd:string"
19         }, [...]
```

The composition between the Library API and the Google Transit API cannot be performed directly because the first API returns the mail address of a library in text format, while the Google Transit API gets geographic coordinates as input. For this reason, between the two compositions a third API

(Google Maps geocoding API) is used to perform geocoding (rule 4). Listing 1.5 shows the annotations of the Google geocoding API.

Listing 1.5. Extract from the description of Google geocoding API.

```
"get": {
    "parameters": [
        { "name": "address" }
    ],
    "semanticAnnotations": {
        "address": {
            "@id": "dbp:address",
            "@type": "xsd:string"
        },
    }
},
"responses": {
    "200": {
        "location": {
            "properties": {
                "lat": { "type": "number" },
                "long": { "type": "number" }
            },
            "semanticAnnotations": {
                "lat": {
                    "@id": "dbp:latitude",
                    "@type": "xsd:float"
                },
                "long": {
                    "@id": "dbp:longitude",
                    "@type": "xsd:float"
                }, [...]
```

Now that all the information on the different ways to get access to the textbook have been collected, we can compose the results to deliver the requested report to the user.

Dealing with *merge composition*, we need to verify the semantic compatibility of at least two different outputs (Figure 4).

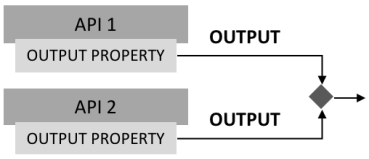

Fig. 4. Schema of merge composition.

To evaluate semantic compatibility in the merge composition, we can define an additional rule:

Rule 5: concepts as unique identifiers. If two or more descriptions share compatible concepts (i.e., they are linked by properties like *owl:sameAs*, *owl:equivalentClass*, *rdfs:subClassOf*, or *rdfs:subPropertyOf*), and these concepts uniquely identify the represented resources (e.g., ISBN for a book, VAT ID for a company, BARCODE for a products), then the outputs of the APIs can be merged.

The Listing 1.6 is a fragment of the Archive.org API[15] description; as shown in line 10, 14, 18, respectively the annotation of the output properties, ISBN, title, author; it is possible to observe how these properties are compatible with the response of Google Books API (Listing 1.1). According to rule 5, the merge composition can occur if compatible properties allow us to conclude that outputs refer to the same resources. In the use case, the *ISBN* can be adopted as unique identifier for books, thus allowing composition of outputs into the final comprehensive report.

Listing 1.6. Extract from the output part of the description of the Archive API.

```
"200": {
    "Book": {
        "type": "object",
        "properties": {
            "ISBN": { "type": "string" },
            "title": { "type": "string" },
            "author": { "type": "string" }, [...]
        },
        "semanticAnnotations": {
            "ISBN": {
                "@id": "dbp:isbn",
                "@type": "xsd:integer"
            },
            "title": {
                "@id": "dbp:title",
                "@type": "xsd:string"
            },
            "author": {
                "@id": "dbp:author",
                "@type": "xsd:string"
            }, [...]
```

5 AutomAPIc: composition of REST APIs

AutomAPIc is a comprehensive tool to manage semantic descriptions and input/ouput composition of services. In this paper, we concentrate on the the description editor, which supports semi-automatic creation of semantic descriptions, and automatic composer, which supports compatibility matching. AutomAPIc is available via Git repository[16]. The Figure 5 shows the architecture of the tool.

It is possible to identify 6 main components: (i) Description Editor, for the definition and management of API descriptions in OpenAPI format; (ii) Description Annotator, for adding semantic annotations; (iii) Composition Editor, which allows for the selection of a set of composable APIs by the user; (iv) API Connector, component for automatic identification of the composable APIs in relation to the composition rules described above; (v) Ontology Connector, component to extract semantic relations by queries to the LOD Cloud[17] with SPARQL query; (vi) Composer API, for the execution of the composition previously defined by the user.

[15] http://blog.archive.org/developers/
[16] https://bitbucket.org/disco_unimib/automapic-tool/
[17] http://lod-cloud.net

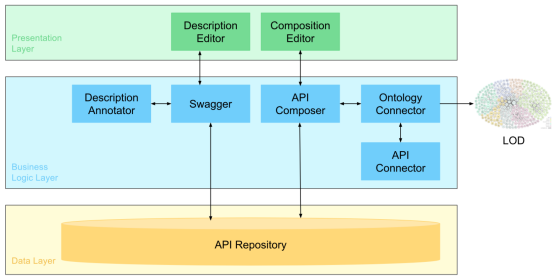

Fig. 5. Architecture of AutomAPIc tool.

5.1 Getting OpenAPI descriptions

The description process is semi-automatically managed by augmenting existing API descriptions, which can be retrieved from existing repositories (e.g., ApisGuru, SmartAPI), or created manually using the Description Editor. These descriptions are represented in JSON or YAML format, and include all relevant information such as available HTTP operations, the list of input parameters and output responses for each operation. The process of creating a description is detailed in Algorithm 1.

Algorithm 1: Retrieve or create API description.

Result: API description

```
if description is available then
    retrieve description from existing repositories and registries of services;
else
    create it manually using the Description Editor;
```

5.2 Adding semantic annotation

If semantic annotations are missing, we need to annotate input and output data. To annotate output data, AutomAPIc provides users with a service that collects a set of output values of GET calls into a table and applie Semantic Table Interpretation [17] techniques to *understand* such values and identify the annotations to be added.

Table interpretation consists of associating data with semantic concepts in an ontological structure, within the LOD Cloud, which aims to represent the knowledge of a certain domain through the connections that exist between these same elements. The GET method is mainly considered since it is the most frequent. In this way API's parameters and properties can be managed by a computer. The code related to the Table Interpretation technique used in this proposal is available through a Git repository[18].

The input parameters are annotated differently because it is not possible to transform the parameters into a table. AutomAPIc provides a service based on

[18] https://bitbucket.org/disco_unimib/mantistable-tool/

Natural Language Processing [2] techniques. In particular the *Stanford CoreNLP* tools[19] [9] has been adopted. These tools provide several libraries that allow for the extraction of entities from API descriptions, which will then be associated with concepts. The application of these techniques on hundred descriptions from the repository *APIs.guru* led to the correct identification of entities and properties for 93% of the cases. Algorithm 2 defines the process to insert semantic annotations in API descriptions. This algorithm revises and extends the one presented in [3].

Algorithm 2: Create and add semantic annotation to API descriptions.

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

5.3 Performing automatic composition

The presence of semantic annotations allows the automatic identification of the composable APIs given a starting API. The API composer component automatically shows the compatible APIs. The possible combinations have been previously calculated by the API connector, through the use of SPARQL queries, in order to apply the compatibility rules (Algorithm 3).

[19] https://stanfordnlp.github.io/CoreNLP/index.html

Algorithm 3: Identification of compatibility between the APIs.

```
Result: Composed APIs
inserting a new API into the system;
parsing of the description;
extraction of semantic annotations;
foreach APIs do
    creation and execution of SPARQL queries to identify the relationships between the
      annotations of the APIs;
    update the graph of possible compositions;
```

6 Validation

To verify the validity of the proposed composition approach, we collected a set of APIs (Table 2) for the creation of a *benchmark* with characteristics that cover all possible cases. The chosen APIs comes from various domains, including public transport, films, books, music and events.

Table 2. Validation dataset.

API	***Description***	***Source***
GEOCODING	Converts an address into latitude and longitude	Google Maps
MARINE CONDITION	Forecast of marine conditions	World Weather Online
WEATHER FORECAST	Weather forecasts	Weather Underground
PHOTOS	Photos geolocated in a specific position	Flickr
NEWS	List of news	NewsAPI
BOOK	List of information about a book	Google
MOVIE	List of information about a film	OMDb API
POI	Points of interest of a city	Sygic API
LIBRARY	List of information about the availability of a book	Opac Unimib
E-COMMERCE	Information regarding the price of a product	Amazon Market
FREE EBOOK	Information on the presence of a free eBook	Archive.org
PLAYLIST	List of songs contained in a playlist	Spotify
LYRICS	Text of a song	Musixmatch API
FLIGHTS	Airport information	Ryanair API
BIKE SHARING	List of bicycles available	City Bike
EVENTS	List of events in a city	EventiFul
HOTEL BOOKING	List of hotels available on a specific date on a certain day	HotelsCombined API
REVIEWS	List of reviews of places and events	TripAdvisor Content API
PUBLIC TRANSPORT	List of information about public transport in a particular place	Google Transit
RESTAURANTS	List of restaurants in a specific city	Zomato API

In a second phase the descriptions and their annotations were analyzed, to identify the possible compositions. Through the combinatorial calculation it is possible to calculate the maximum number of combinations. In particular, given 20 APIS, using provisions without repetitions (since an API cannot be composed with itself), the maximum number of compositions is 380.

As shown in Figure 6, depending on parameters and annotations, the actual combinations are twenty four. AutomAPIc was able to identify the 85% of them. Table 3 reports the *confusion matrix* of the results, where attributes are: (i) TP: number of correctly composed APIs, (ii) FP: number of APIs that were composed but which should not be composed, (iii) FN: number of APIs that were not composed but that had to be composed, (iv) TN: number of APIs that were not to be composed and were not composed. The *accuracy* of the system is

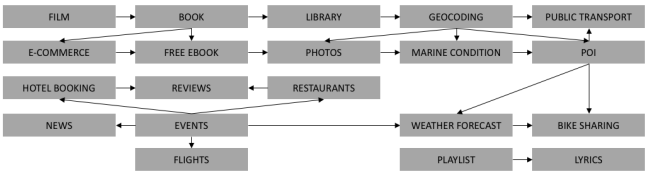

Fig. 6. List of the possible compositions.

$(TP+TN)/Total = 0.99$. Going into detail, the combinations that led to composition failures are mainly three: weak support to manage concepts connected by the *owl:subProperty* relation, incomplete relationships between ontologies (e.g., DBpedia and KBpedia), and inaccurate semantic annotations of parameters returned by table interpretation techniques. A discussion on the quality of results of table interpretation techniques is out of scope of this paper, however interested readers can refer to [17] for details.

Table 3. Confusion matrix.

Tot. = 380	Composed	Not - Composed
Composed	**TP = 17**	**FP = 0**
Not - Composed	**FN = 3**	**TN = 360**

7 Conclusions and future work

The work presented in this paper aims to propose an extension of the OpenAPI specification to support the semantic annotations of services descriptions and the automatic composition of services. The goal is to support users without specific skills to manage semantics annotations, thus encouraging the delivery of semantically annotated descriptions. For this reason, two solutions have been proposed. For the annotation of input parameters, the use of Natural Language Processing (NLP) techniques has been proposed, while for the annotation of output properties, a reviewed Table Interpretation approach has been developed. The validation of the proposal through a subset of real APIs has underlined how the use of semantic annotations and the definition of a set of composition rules lead to an effective support to the composition of APIs, even if further development is necessary to improve both precision and recall. Future work will go in that direction to consolidate the AutomAPIC tool, along with fully integration with the Swagger interface. Moreover, further investigations will be conducted to verify the quality of the table interpretation outputs, which play an important role in our composition approach. In addition, a user-centric evaluation is planned in order to verify the ability of users to manage this new type of descriptions with semantic annotations. Finally, to enhance the automation of the entire process, we will study how to capture and model the user requirements.

References

1. Chinnici, R., Moreau, J.J., Ryman, A., Weerawarana, S.: Web services description language (wsdl) version 2.0 part 1: Core language. W3C recommendation 26, 19 (2007)
2. Chowdhury, G.G.: Natural language processing. Annual review of information science and technology 37(1), 51–89 (2003)
3. Cremaschi, M., De Paoli, F.: Toward automatic semantic api descriptions to support services composition. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) Service-Oriented and Cloud Computing. pp. 159–167. Springer International Publishing, Cham (2017)
4. Euzenat, J., Shvaiko, P.: Ontology Matching. Springer-Verlag, Berlin, Heidelberg (2007)
5. Gomadam, K., Ranabahu, A., Sheth, A.: Sa-rest: semantic annotation of web resources. W3C Member Submission 5, 52 (2010)
6. Kopeckỳ, J., Vitvar, T., Fensel, D., Gomadam, K.: hrests & microwsmo. STI International, Tech. Rep. (2009)
7. Lausen, H., Farrell, J.: Semantic annotations for wsdl and xml schema. W3C recommendation, W3C 69 (2007)
8. Lucky, M.N., Cremaschi, M., Lodigiani, B., Menolascina, A., De Paoli, F.: Enriching api descriptions by adding api profiles through semantic annotation. In: Proc. of the 14th ICSOC 2016. pp. 780–794. LNCS Springer (2016)
9. Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., McClosky, D.: The stanford corenlp natural language processing toolkit. In: Proceedings of 52nd Annual Meeting of the Association for ComputationalLinguistics: System Demonstrations. pp. 55–60. Association for Computational Linguistics (2014)
10. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., et al.: Owl-s: Semantic markup for web services. W3C member submission 22, 2007–04 (2004)
11. Paulraj, D., Swamynathan, S., Madhaiyan, M.: Process model-based atomic service discovery and composition of composite semantic web services using web ontology language for services (owl-s). Enterprise Information Systems 6(4), 445–471 (2012)
12. Rao, J., Su, X.: A survey of automated web service composition methods. In: Cardoso, J., Sheth, A. (eds.) Semantic Web Services and Web Process Composition. pp. 43–54. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
13. Roman, D., Kopeck, J., Vitvar, T., Domingue, J., Fensel, D.: Wsmo-lite and hrests: Lightweight semantic annotations for web services and restful apis. Web Semantics: Science, Services and Agents on the World Wide Web 31, 39 – 58 (2015)
14. Sheng, Q.Z., Qiao, X., Vasilakos, A.V., Szabo, C., Bourne, S., Xu, X.: Web services composition: A decades overview. Information Sciences 280, 218 – 238 (2014)
15. 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)
16. Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.F.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More. Prentice Hall PTR, Upper Saddle River, NJ, USA (2005)
17. Zhang, Z.: Effective and efficient semantic table interpretation using tableminer+. Semantic Web 8(6), 921–957 (2017)