Instantiating REST Services to the OpenAPI Ontology

Nikolaos Mainas, Aikaterini Karavisileiou, Fotios Bouraimis, and Euripides G.M. Petrakis

School of Electrical and Computer Engineering, Technical University of Crete (TUC)
Chania, Crete, Greece
{akaravisileiou, fbouraimis}@isc.tuc.gr, {nmainas, petrakis}@intelligence.tuc.gr

Abstract. The proposed work is an attempt to contribute towards understanding the meaning of Web services using ontologies, a well established technology for representing knowledge in a domain (in this work, in the Web services domain). OpenAPI Specification (OAS) is a description format for REST APIs. In order for a machine to understand the meaning of REST services, OpenAPI service descriptions must be unambiguous so that, the services can be searched, discovered and used by other services. In a previous work we analysed the reasons that cause ambiguities in OpenAPI descriptions and showed that, in order to eliminate ambiguities, OpenAPI properties must be semantically annotated and mapped to a semantic model. Leveraging latest results for hypermedia-based construction of Web APIs (i.e. Hydra) and the latest update of the standard (OpenAPI v3.0), the present work proposes a reference ontology for REST services along with a formal procedure for converting OpenAPI service descriptions to instances of this ontology.

Keywords: Web service, Ontology, OpenAPI, REST, SHACL, Hydra

1 Introduction

The increasing interest in Web service architectures over the past years has led to the proliferation of Web service offerings over the Internet. Consequently, the need for efficient and accurate service discovery based on user needs has become a significant challenge. Typically, Web services are described in plain text which the users have to browse and read, in order to determine whether a service meets their needs. However, text descriptions of Web services are not readable by machines and in some cases are inaccurate or vague. Web services need to be formally described in a way that is understandable by both, humans and machines. The last requirement would not only improve the accuracy of service descriptions but also, would allow for services to be discovered by other services and be orchestrated in composite services or applications.
The need for standardizing technologies for service publishing and discovery is of crucial importance for their adoption and market success. OpenAPI Specification (OAS) is a widely adopted standard for describing REST APIs which is supported by large industry users like Google, Microsoft, IBM, Oracle and many others. OAS 3.0 is the first major update of the specification since 2015. OAS 3.0 features a more elaborate (yet simple) structure and format than its predecessor OAS 2.0. The requirement for a single host server is relaxed (allowing a service to be installed on multiple servers). A request body is more flexible and allows consumption of different media types, such as JSON, XML, HTML, plain text and others. The descriptions for parameters have changed: FormData parameter was removed and, the cookie parameter type was introduced for documenting APIs that use cookies. The definition of Schema objects is enhanced with additional properties (e.g. anyOf, oneOf, not) allowing for the creation of more complex schemas with various data types. Regarding security definitions, OpenAPI v3.0 is enhanced with support for OpenID Connect. OpenAPI 3.0 now features a Components field where various reusable objects can be defined (i.e. responses, parameters, headers, links, callbacks, schemas and security schemes).

Two new features are introduced referred to as LINKS and CALLBACKS. LINKS are defined in the service response section to allow values returned by a service call to be used as input for a next call. This is an attempt of OpenAPI 3.0 to incorporate HATEOAS functionality in the specification. HATEOAS is a unique property of REST architectural style that allows a client (i.e. an application) to discover resources by navigating through hyperlinks provided by service responses. Finally, CALLBACKS is a feature for defining asynchronous APIs or Webhooks. CALLBACKS define the requests that the described service will send to another service in response to certain events. An application of this feature would be for describing publish-subscribe mechanisms which allow services to publish information and other services subscribing to them to get notified when this information becomes available.

OpenAPI format is based on JSON (or YAML) and a large set of properties are available for composing service descriptions. Despite its user friendliness, OpenAPI 3.0 service descriptions can be vague: the same property may appear with different names within the same OpenAPI document or, its meaning may not be defined at all. OpenAPI does not provide a mechanism for detecting or for dealing with ambiguities. In our recent work we analyzed the causes of ambiguity and we concluded that, in order to eliminate these ambiguities, OpenAPI properties must be semantically annotated and associated to entities of a semantic model (e.g. in www.schema.org shared vocabulary). Then, we showed that is plausible to represent OpenAPI descriptions using ontologies and we proposed a reference ontology for OpenAPI service descriptions. This is the

1 https://www.openapis.org
2 https://blog.restcase.com/6-most-significant-changes-in-oas-3-0/
3 https://openid.net/connect/
4 https://restfulapi.net/hateoas/
first work in the literature that handles ambiguities in OpenAPI version 3.0 descriptions.

The present work forwards this approach in certain ways: (a) presents the Beta (stable) version of the OpenAPI ontology and (b) proposes a formal methodology and an algorithm for converting OpenAPI version 3.0 service descriptions to instances of the ontology. A related Web application\footnote{http://www.intelligence.tuc.gr/semantic-open-api/} which supports uploading of OpenAPI descriptions of REST services, instantiation to the ontology and downloading of the resulting ontology, is available on the Web servers of our laboratory for evaluation by all users. The new (Beta version) of the ontology takes full advantage of Hydra\footnote{https://www.w3.org/TR/shacl/} and SHACL. Hydra is a promising technology towards understanding and constructing Web services that meet the HATEOAS requirement of REST architectural style. OpenAPI 3.0 ontology incorporates features of Hydra for modeling service operations along with models not foreseen in Hydra (e.g. REST security features, headers, constraints). Classes together with constraints on class properties are described using SHACL allowing for service descriptions to be validated against the ontology. The OpenAPI ontology combines the advantages of all and supports the efficient representation and dynamic discovery of hypermedia driven APIs.

Instantiating OpenAPI descriptions and services is a rather complicated process. The reasons can be any or all the following: the same property may appear many times in the same document (with the same or different meaning), with different scopes (local property declarations overshadow global ones) or, it can be nested inside other properties. In addition, a Schema object can be defined as an extension or the composition of existing models (e.g. using the allOf property) which add extra complexity to the instantiation process. The proposed algorithm handles all these issues.

The ontology should not be necessarily visible to the end-user or API developer. It acts in the background and its role is twofold: (a) informs the user for possible ambiguities and inconsistencies in OpenAPI descriptions and (b) works as a service repository and for discovering similarities between Web APIs (e.g. in the spirit of Hamza et al.\footnote{http://www.intelligence.tuc.gr/semantic-open-api/} for discovering OpenAPI compliant REST APIs). Instantiating REST services to the OpenAPI ontology would enable application of query languages (e.g. SPARQL) for service discovery, or reasoning for detecting inconsistencies in service descriptions and also, enable service orchestration and synthesis of composite services. The last, is a long and more ambitious goal of our approach.

Related work on services description languages (with emphasis to REST services) is presented in Sec.\footnote{http://www.intelligence.tuc.gr/semantic-open-api/} 2. The OpenAPI 3.0 approach is discussed in Sec.\footnote{https://www.w3.org/TR/shacl/} 3. The instantiation of service descriptions to the OpenAPI 3.0 ontology is discussed in Sec.\footnote{http://www.intelligence.tuc.gr/semantic-open-api/} 4 followed by issues for future work in Sec.\footnote{https://www.w3.org/TR/shacl/} 6.
2 Related Work

OpenAPI is a fairly new technology and despite its high impact and adoption by the industry, it has not been explored in depth in the related (Web services) literature. As a result, the proposed model is also novel and the related work is very limited.

Syntactic description languages describe the requirements for establishing a connection with a service and the message formats to successfully communicate with it. WSDL\(^7\) for SOAP, WADL\(^8\) for REST, are popular mainly due to their simplicity and compatibility with common machine readable formats like XML and JSON. They are not specifically designed for hypermedia-driven APIs (such as REST) that call for the dynamic discovery of resources at run-time (referred to as HATEOAS). Their reputation has been overshadowed by OpenAPI Specification\(^10\) formerly known as Swagger.

Semantic approaches build-upon the idea of describing services by means of semantic models (i.e. ontologies) and are more capable of supporting automated service discovery and composition. WSMO\(^11\) defines a conceptual model and WSML language for the semantic description of Web services. OWL-S\(^12\) is an upper ontology for Web services but, similar to all other methods, do not support the dynamic discovery of resources at runtime.

Hydra\(^3\) simplifies the construction of hypermedia-driven APIs. The Hydra vocabulary defines concepts that a server can use to advertise valid state transitions to a client as a result of a sequence of service invocations. Server responses are provided as JSON-LD which a client can use at run-time to discover the available actions and resources, in order to formulate new HTTP requests and achieve a specific goal. Hydra is a promising technology towards understanding and constructing Web services that meet the HATEOAS requirement of REST architectural style.

OAS 2.0. Musyaffa et al.\(^5\) introduce annotations in Schema and Parameter objects for OpenAPI 2.0. They do not handle all causes of ambiguity nor do they handle OpenAPI 3.0 descriptions. The annotations appear within text properties and cannot be interpreted by a machine without pre-processing. Schwichtenberg, Gerth and Engels\(^6\) map OpenAPI 2.0 service descriptions to OWL-S ontology but do not deal with any causes of ambiguity in service descriptions, nor do they handle security properties. Their approach attempts to find a mapping of OAS 2.0 Schema objects to OWL-S using heuristics and name similarity matching techniques. The mapping is error prone and need to be adjusted manually. Most important, their choice of OWL-S model for representing REST services is controversial: OWL-S is good for SOAP services but not good for

\(^7\) https://www.w3.org/TR/wsd1.html
\(^8\) https://www.w3.org/Submission/wadl/
\(^9\) https://raml.org
\(^10\) https://www.openapis.org
\(^11\) https://www.w3.org/Submission/WSMO/
\(^12\) https://www.w3.org/Submission/OWL-S/
hypermedia-driven Web services like REST. Finally, they do not handle OAS 3.0 descriptions.

In a recent contribution, Hamza et al. [4] propose a example-driven approach and a representation of REST APIs for discovering OpenAPI compliant REST Web APIs. This facilitates the API discovery favoring software reuse. The approach does not deal with ambiguity in OpenAPI and is complementary to our work, in the sense that, our proposed ontology representation is a far more powerful tool for supporting services discovery enhanced with reasoning for service synthesis and integration of existing APIs.

3 Annotated OpenAPI Descriptions

Fig. 1 illustrates the structure of an OAS 3.0 service description[13]. OAS 3.0 service descriptions comprise of many parts (objects). Each object has a list of properties which can be objects as well (this way, objects can be linked to each other). The Info object provides non-functional information such as the name of the service, service provider, license information and terms of the service. The Servers object provides information on where the API’s servers are located. The Servers object specifies the base-path (the part of the URL that appears before the endpoint) of an API request. There are also variables that can be populated at run-time. Servers can be defined for different operations (i.e. a Servers object can be added as property in the Path object of an Operation object). If defined, these locally declared servers will override the base (i.e. global) servers.

![OpenAPI 3.0](https://blog.readme.io/an-example-filled-guide-to-swagger-3-2/)

**Fig. 1.** OpenAPI document structure.

The Security requirement object lists the security schemes that the service applies prior to executing an operation. Its name must correspond to a security scheme declared in Security Schemes under the Components object. The specification offers support for basic HTTP authentication, API keys, OAuth2 common flows or grants (i.e. ways of retrieving an access token) and OpenID Connect.

The Paths object describes the available operations (i.e. HTTP methods) and contains the relative paths for the service endpoints (which are appended to a server URL in order to construct the full URL of an operation).

The Components object holds a set of reusable objects which can be responses, parameters, schemas, request bodies, security schemes and more. The Schemas object define data structures that are used to describe the request and response messages. A Schema object can be a primitive (string, integer), an array or a model. A Schema object may also have properties of its own accord (i.e. externalDocs) and properties supporting model composition and polymorphism. The discriminator property in a schema informs about the existence of an alternative schema (based on a keyword such as oneOf, anyOf, allOf).

The Responses object describes the expected responses of an operation, by mapping them to a specific HTTP status code. A response object defines the message content, as well as HTTP headers that a response may contain. Parameters object describes parameters that operations may use. The specification, categorizes parameters into:

- Path parameters, used in cases where the parameter values are part of operation’s path.
- Query parameters, appended to the url when sending a request.
- Header parameters, defining additional custom headers that may be sent in a request.
- Cookie parameters, passed in the Cookie header.

OpenAPI is typically discussed with references to Swagger Petstore\textsuperscript{16} service example providing information about pets and clients placing orders to a pet shop. In this paper, it is plausible to discuss the extensions to the model using the same example. The original example has been modified to include additional properties showing polymorphism, security properties as well as the use of all extension properties \textsuperscript{7}.

In an OpenAPI document there are elements that share the same meaning that a human may understand but a machine cannot. In our recent work \textsuperscript{1,2} we introduced extra properties to annotate OpenAPI properties which are ambiguous. Table \textsuperscript{1} summarizes the extension properties, their scope and their meaning. The x-refersTo extension property specifies the association between an OpenAPI element and a concept in a semantic model. Listing \textsuperscript{1.1} is an excerpt from the Swagger Petstore\textsuperscript{16} service description (widely used as an ex-

\textsuperscript{14} https://oauth.net/2/
\textsuperscript{15} https://openid.net/connect/
\textsuperscript{16} https://github.com/OAI/OpenAPI-Specification/blob/master/examples/v3.0/petstore.yaml
\textsuperscript{17} http://petstore.swagger.io/
Table 1. OAS extension properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Applies to</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x-refersTo</td>
<td>Schema Object</td>
<td>The concept in a semantic model that describes an OAS element.</td>
</tr>
<tr>
<td>x-kindOf</td>
<td>Schema Object</td>
<td>A specialization between an OAS element and a concept in a semantic model.</td>
</tr>
<tr>
<td>x-mapsTo</td>
<td>Schema Object</td>
<td>An OAS element which is semantically similar with another OAS element.</td>
</tr>
<tr>
<td>x-collectionOn</td>
<td>Schema Object</td>
<td>A model describes a collection over a specific property.</td>
</tr>
<tr>
<td>x-onResource</td>
<td>Tag Object</td>
<td>The specific Tag object refers to a resource described by a Schema object.</td>
</tr>
<tr>
<td>x-operationType</td>
<td>Operation Object</td>
<td>Clarifies the type of operation.</td>
</tr>
</tbody>
</table>

ample in all references to OpenAPI). The x-refersTo extension property is used to semantically annotate a Pet model and its properties: it associates the model with Pet class in Product ontology.\(^{18}\) If the Pet model describes a specific group of pets (e.g. dogs), the x-kindOf extension property is used instead to denote that the model is a subclass of the referred semantic concept. The x-mapsTo is used to define Schema object elements within the same OpenAPI document that share the same semantics. In Listing 1.1 the x-mapsTo property is used in Parameters object to dictate that query parameter name refers to Pet.name in Schemas object. There is a semantic similarity between the id property of Pet model and the petId property in https://schema.org/serialNumber which later appears in the document as petId (not shown in the example). Similarly, property Pet should refer to http://www.productontology.org/doc/Pet and name should refer to http://purl.org/goodrelations/v1#name. A human may easily infer these semantic similarities, either by the element names or by the description that may be provided. However, in order for a machine to act similarly to a human it is necessary to provide additional information, that clarifies these relations.

Listing 1.1. OpenAPI model polymorphism example.

parameters:

...  
Query:
  name: name
  in: query
  required: true
  schema:
    type: string
    x-mapsTo: ‘#/components/schemas/Pet.name’

schemas:
  Pet: # A Pet model extended with SOAS 3.0 properties

\(^{18}\) http://www.productontology.org/
type: object
x-refersTo: http://www.productontology.org/doc/Pet
properties:
  name:
    type: string
    x-refersTo: http://purl.org/goodrelations/v1#name
  photoUrls:
    type: string
    x-refersTo: https://schema.org/image
  id:
    type: integer
    x-refersTo: https://schema.org/serialNumber
required:
  - name
  - photoUrls
discriminator:
  propertyName: name

Dog: # A Dog model extending the Pet Model
x-kindOf: http://www.productontology.org/doc/Pet
allOf:
  - $ref: '#/components/schemas/Pet'
  - type: object
    properties:
      breed:
        type: string
        required:
          - breed

Cat: # A Cat model extending the Pet Model
x-kindOf: http://www.productontology.org/doc/Pet
allOf:
  - $ref: '#/components/schemas/Pet'

The x-collectionOn extension property is used to indicate that a model in Schemas object is actually a collection. Typically, a collection (or a list) of resources in OpenAPI 3.0 is described using the array type. However, often the definition of collection, is encapsulated within an object type with additional properties. Then, x-collectionOn property is used to denote the data types of the objects of the collection. Listing 1.2 defines a model as a collection of Pet objects (totalItems property denotes population).

Listing 1.2. Model definition representing a collection.

... schemas:
  PetCollection: # A Pet Collection definition
    x-collectionOn: pets
The `x-onResource` extension property is used in Tag Objects to specify the resource that a tag refers to. In OpenAPI 3.0, tags are used to group operations either by resources or, by any other qualifier. If the tag is used to group operations by resources, a human may recognize that the referred resource is described by a Schema object in Schemas object but, a machine cannot. The `x-onResource` property is used to associate the tag with a Schema object that describes a specific resource. In Listing 1.3, `x-onResource` property is assigned on a pet tag that provides information regarding the operations that are available for Pet model in Schemas object.

Listing 1.3. Excerpt from Swagger Petstore service description.
Finally, $x$-operationType extension property is used to specify the type of an Operation object. A request is characterized by the HTTP method it applies. However, the semantics of the HTTP methods are too generic and may have a more specific meaning. For example, this property can be used to clarify that a get request can actually mean a search operation on Schema objects. In Listing 1.3, this property is used to clarify that a get request on path /pet/findByStatus is a search operation on pets based on their status. The value of the property is a URL pointing to the concept that semantically describes the operation type. The Action type of the Schema.org vocabulary provides a detailed hierarchy of Action sub-types that can be used by the property.

4 OpenAPI Ontology

The OpenAPI ontology in Fig. 2 captures all information in a Semantic OpenAPI description. Properties of classes are mapped to classes as well. At the heart of the ontology is Hydra Core Vocabulary, enhanced with additional models in order to capture information about security, headers and constraints.

Document class provides general information (Info class) regarding the service and specifies service paths, the entities and the security schemes that it supports. Path class represents (relative) service paths (pathName property). Operation class provides information for issuing HTTP requests. Request bodies are represented by Body class, while responses are declared in Response class specifying the status code and the data returned. The entire range of HTTP responses is represented. The MediaType class describes the format (the most common being JSON, XML) of a request or response body data. Class Operation refers to a security scheme in SecurityRequirement class.

Fig. 3 shows the security schemes supported by OpenAPI. Class Security has security schemes as sub-classes. Class OAuth2 has different flows (grants) as sub-classes. If the security scheme is of type OAuth2 or OpenID Connect, then scope names are defined as properties.

Listing 1.4 illustrates how an OAS Path item and Operation are defined in the ontology using the example of Listing 1.3. Class Security defines the security schemes that the specification supports. Class Operation refers to a security scheme using class SecurityRequirement, which in the case of the OAuth2 security scheme represents the scopes of the operation. Operation path2_op1 refers to a SecurityRequirement individual, specifying an OAuth2 security scheme (i.e. petstore_oauth individual) and the corresponding scope (i.e. read_pets and write_pets individuals). Individual path2_op1 is also considered to be an individual of SearchAction type defined in Schema.org vocabulary (i.e. as defined by the x-operationType extension property).

19 http://schema.org/Action
Fig. 2. OpenAPI 3.0 ontology.

Listing 1.4. Representation of Path and Operation.

```plaintext
... ex:path2
  a openapi:Path;
  openapi:pathName "/pets/findByStatus";
ex:path2_op1
  a openapi:Operation, schema:SearchAction;
openapi:onPath ex:path2
openapi:method openapi:GET;
openapi:tag ex:tag_pet;
openapi:parameter ex:query_status;
openapi:response [
  openapi:statusCode 200;
  openapi:content [
```

openapi:mediaTypeName "application/json";
openapi:schema ex:PetCollectionShape
]
openapi:response [ openapi:statusCode 400;
openapi:description "Invalid status value".
]
openapi:security [ openapi:securityType ex:petstore_oauth;
openapi:scope ex:read_pets, ex:write_pets
]; openapi:name "findPetsByStatus";
openapi:summary "Finds Pets by Status"
openapi:description "Multiple status values with coma seperated strings".
...

Schema objects are expressed as classes, object and data properties using SHACL. SHACL is an RDF vocabulary that can be used to define classes together with constraints on their properties. It provides built-in types of constraints (e.g. cardinality: minCount, maxCount) and allows expression of constraints on the type of properties and on the values they can take. Table 2 shows the mapping between Schema object properties and SHACL. Schema objects are mapped to Shape class. The NodeShape class defines the properties of a class and specifies whether a class may contain additional properties (additionalProperties) of a specific type. Additionally, it represents operations related to a class (supportedOperation), which come from x-onResource extension property. Class PropertyShape represents the properties of a class, their datatype and restric-
Table 2. Mapping OpenAPI Schema Object properties to SHACL.

<table>
<thead>
<tr>
<th>Schema Object property</th>
<th>SHACL property</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum</td>
<td>sh:exclusiveMaximum if openAPI exclusiveMaximum is true</td>
</tr>
<tr>
<td>exclusiveMaximum</td>
<td>sh:inclusiveMaximum if openAPI exclusiveMaximum is false</td>
</tr>
<tr>
<td>minimum</td>
<td>sh:inclusiveMinimum if openAPI exclusiveMinimum is true</td>
</tr>
<tr>
<td>exclusiveMinimum</td>
<td>sh:inclusiveMinimum if openAPI exclusiveMinimum is false</td>
</tr>
<tr>
<td>maxLength</td>
<td>sh:maxLength</td>
</tr>
<tr>
<td>minLength</td>
<td>sh:minLenght</td>
</tr>
<tr>
<td>pattern</td>
<td>sh:pattern</td>
</tr>
<tr>
<td>maxItems</td>
<td>sh:maxCount</td>
</tr>
<tr>
<td>minItems</td>
<td>sh:minCount</td>
</tr>
<tr>
<td>enum</td>
<td>sh:in</td>
</tr>
<tr>
<td>allOf</td>
<td>sh:and</td>
</tr>
<tr>
<td>oneOf</td>
<td>sh:xone</td>
</tr>
<tr>
<td>anyOf</td>
<td>sh:or</td>
</tr>
<tr>
<td>not</td>
<td>sh:not</td>
</tr>
<tr>
<td>default</td>
<td>sh:defaultValue</td>
</tr>
</tbody>
</table>

The SHACL class PetShape is defined according to Schema object definition of Pet with the addition of new data properties and constraints (e.g. each pet has exactly one name and an id). A Schema object defined using the combination of allOf property and a discriminator property, is represented as a subclass of the

Listing 1.5. Representation of an OpenAPI model.

```
ex:PetShape
  a sh:NodeShape ;
  sh:targetClass po:Pet ;
  sh:property [
    sh:path purl:name ;
    sh:name "name" ;
    sh:datatype xsd:string ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ];
  sh:property [
    sh:path schema:serialNumber ;
    sh:name "id" ;
    sh:datatype xsd:integer ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] .
```

Listing 1.5 shows how the Pet model of Listing 1.1 is represented in the ontology. The model contains references to Schema.org vocabulary using x-refersTo. The SHACL class PetShape is defined according to Schema object definition of Pet with the addition of new data properties and constraints (e.g. each pet has exactly one name and an id). A Schema object defined using the combination of allOf property and a discriminator property, is represented as a subclass of the
A request or response body (defined using content property) is used to send and receive data via the REST API respectively (a response contains also a response code e.g. 200, 400 etc.). Media type is a format of a request or response body data in different formats, the most common being JSON, XML, text and images. They are typically defined in Paths object; however re-usable bodies can also be defined in Components object. Each media type includes a Schema property, defining the data type of the message body. Request and response bodies are represented as properties of class Operation. Request and Response bodies in particular, are defined as classes and so is defined their media type. Class Encoding defines keywords denoting serialization rules for media types with primitive properties (e.g. contentType for nested arrays or JSON).

5 Instantiating OpenAPI Descriptions

Details on the algorithms presented here and their implementation can be found in the author’s thesis [7]. Algorithm [1] scans the OpenAPI document of a service.
and instantiates OpenAPI objects to classes of the ontology. In particular, after uploading the ontology in the memory, the algorithm will scan the OpenAPI file to extract info, servers, securitySchemes, securityRequirements, tags and paths objects. These objects will become individuals of their corresponding classes.

**Algorithm 1** Instantiating OpenAPI object to ontology.

```
1: procedure PARSEDOCUMENTOBJECT(OpenAPI doc) ▷ Main body of algorithm:
2:     ont = InitializeOntologyModel();
3:     INFOOBJECT(ont, doc);
4:     Read servers property of OpenAPI object;
5:     globalServerList = SERVEROBJECT(ont, servers);
6:     SECURITYSCHEMEOBJECT(ont, doc);
7:     Read security property of OpenAPI object;
8:     globalSecurityReqList = SECURITYREQUIREMENTOBJECT(ont, security);
9:     tagShapeMap = TAGOBJECT(ont, doc);
10: PATHOBJECT(ont, doc, globalServerList, globalSecurityReqList, tagShapeMap);
11: function INFOOBJECT(Ontology ont, OpenAPI doc): ▷ Function definitions:
12:     Info object becomes individual of class Info;
13: function SERVEROBJECT(Ontology ont, List servers):
14:     Initialize list containing server individuals (serverList);
15:     for Server object ∈ servers do
16:         Server object becomes individual of class Server;
17:         add individual in serverList;
18:     return serverList
19: function SECURITYSCHEMEOBJECT(Ontology ont, OpenAPI doc):
20:     Each Security Scheme object becomes individual of class Security;
21: function SECURITYREQUIREMENTOBJECT(Ontology ont, List security):
22:     Initialize list containing Security Requirement individuals (securityReqList);
23:     for Security Requirement object ∈ security do
24:         Security Requirement object becomes individual of class SecurityRequirement;
25:         add individual in securityReqList;
26:     return securityReqList
```

The OpenAPI object (the root object of the OpenAPI document) is mapped to class Document. There may exist more than one appearances of servers or securityRequirements in an OpenAPI file. Property servers declares server information which applies across the description (global servers). This will be overwritten by server information defined in Path or Operations objects. Similarly, Security property declares security requirements that are used across the API. Security requirements declared by an operation will also override global declaration of security requirements. Property Tags contains the Tag objects for operations which are grouped. Through the x-onResource property Tag objects can associate operations with Schema objects.
Algorithm 3 illustrates the instantiation of Tag objects in the ontology and how x-onResource relations are handled. In Algorithm 4, Tag names and their associated Shapes are kept in a Map structure (tagShapeMap) that will be used when instantiating Operation objects. tagShapeMap defines a mapping (key, value) between a string and an individual. An entry to the tagShapeMap will contain the tag name and the corresponding Shape individual. This will take place if an x-onResource annotation has been defined on a tag object (i.e. describing a Schema object). The Schema object will be converted to an individual of the Shape class and will be added inside the corresponding mapping (e.g. (tag.getName(), SchemaInd)).

Algorithm 2 Instantiating Path objects in OpenAPI ontology.

```java
1: function PATHOBJECT(Ontology ont, OpenAPI doc, List globalServerList, List globalSecurityReqList, Map tagShapeMap)
2:     for Path Item object ∈ doc do
3:         Create an individual of class Path;
4:         Select the server list that will be used in operations (pathServerList);
5:         if servers property in Path Item object is empty then
6:             pathServerList = globalServerList;
7:         else
8:             Read Server objects of Path Item object (path level);
9:             pathServerList = SERVEROBJECT(ont, servers);
10:        Read Parameter objects of Path Item object (path level);
11:        pathParameterList = PARAMETEROBJECT(ont, parameters);
12:        for Operation object (opObj) ∈ Path Item object do
13:            operationInd = OPERATIONOBJECT(ont, opObj, pathServerList, globalSecurityReqList, pathParameterList, tagShapeMap);
14:            add property onPath to operationInd associating Path and Operation individuals;
15:    function PARAMETEROBJECT(Ontology ont, List parameters) -> Function definition:
16:        Initialize list containing parameter individuals (parameterList);
17:        for Parameter Object ∈ parameters do
18:            Create individual of corresponding class (PathParameter, Query, Cookie, Header);
19:            add individual in parameterList;
20:        return parameterList
```

Algorithm 2 shows how Path objects are converted to individuals of class Path. This is where Operation individuals are created with their respective properties (i.e. tag, security, parameter, server). This is done after creating individuals for Server and Parameter objects that may be declared in a Path. Server objects declared in Path (as stated in OpenAPI Specification) will override global declaration of Server Objects.

Algorithm 4 shows how the individuals of class Operation are created. When the x-operationType annotation is used, the Operation individual is also considered as an individual of the provided Semantic entity (Listing 1.4). The structural elements of the operation (e.g. responseObject, requestBody, security and
Algorithm 3 Create instances of Tag class.

```plaintext
1: function TAGOBJECT(ontology ont, OpenAPI doc)
2:     initialize a Map structure (tagShapeMap) to hold the entries of Tag and Shape individuals;
3:     for Tag Object ∈ doc do
4:         create an individual of class Tag;
5:         if Tag object contains the x-onResource property then
6:             find the Schema object it refers to;
7:             convert Schema Object into an individual of class Shape;
8:             add individuals (Tag and Shape) into tagShapeMap;
9:     return tagShapeMap
```

tag become properties of the Operation individual (e.g. operationInd property:tag tagInd is an example triple). Property method defines the type of an operation (get, put, etc.) which are already defined in the OpenAPI ontology. Parameter objects can be any of type Path, Query Header or Cookie and are instantiated to the corresponding classes (i.e. PathParameter, Query, Header and Cookie). Then, parameter individuals are associated to the Operation individual using the respective properties. Finally, property onPath associates an Operation individual with a Path individual.

5.1 Composition and Polymorphism

The OpenAPI Specification allows the combination and extension of model definitions using the allOf property. However, in order to support polymorphism the discriminator property is used determining which Schema definition validates the structure of the model. A model defined using the allOf property is semantically considered as a subclass of the model that extends. In the Swagger Petstore example any model that extends the Pet model (Listing.1.1) is defined as a subclass of the semantic entity it describes. If Schema objects are not semantically annotated, then the corresponding classes are defined with the respective subclass relation. For example, if Cat and Dog models in Listing.1.1 were not semantically annotated (missing x-kindOf property), then during ontology creation, classes Cat and Dog would be defined and declared as subclasses of the described semantic entity in Pet model. The newly created classes, could also be used for semantically annotating other OpenAPI service descriptions.

OpenAPI service description may have Schema objects sharing common properties. Instead of describing these properties for each Schema repeatedly, these schemas can be described as a composition of common properties and schema-specific properties. These Schema objects should not be related to a semantic entity. To handle these cases we suggest annotating Schema objects with the property x-refersTo: none, so that the algorithm should not attempt to relate these Shapes with a semantic entity.
Algorithm 4 Converting Operation object to instances of class Operation.

1: function OPERATIONOBJECT(Ontology ont, Operation Object opObj, List servers, List securityReqList, List parameters, Map tagShapeMap)
2: Create individual (operationInd) of class Operation;
3: Check for the x-operationType property;
4: if opObj contains the x-operationType property then
5: Add operationInd as member of the class the property refers;
6: for tag ∈ opObj do
7: Add property tag to operationInd;
8: if tag exists in tagShapeMap then
9: Get the corresponding Shape Individual;
10: Add property supportedOperation to Shape individual;
11: if servers property in opObj is not empty then
12: for Server Object ∈ servers do
13: Server object becomes individual of class Server;
14: Add property server to operationInd;
15: else
16: for server individual ∈ pathServerList do
17: Add property server to operationInd;
18: if security property in opObj is not empty then
19: for Security Requirement object ∈ security do
20: Security Requirement object becomes individual of class SecurityRequirement;
21: Add property security to operationInd;
22: else
23: for security requirement individual ∈ globalSecurityReqList do
24: Add property security to operationInd;
25: for Parameter object ∈ parameters property of opObj do
26: Parameter object becomes individual of the corresponding class (PathParameter, Query, Cookie, Header)
27: Add property parameter to operationInd for PathParameter and Query individuals
28: Add property cookie to operationInd for Cookie individuals
29: Add property requestHeader to operationInd for Header individuals
30: for parameter individual ∈ pathParameterList do
31: if parameter individual doesn’t exist in parameters of operationInd then
32: Add property parameter to operationInd for PathParameter and Query individuals
33: Add property cookie to operationInd for Cookie individuals
34: Add property requestHeader to operationInd for Header individuals
35: if requestBody property in opObj then
36: RequestBody object becomes individual of class Body
37: Add property requestBody to operationInd
38: for Responseobject ∈ responsesproperty of opObj do
39: Response object becomes individual of class Response
40: Add property response to operationInd
41: return operationInd
5.2 Use case example

Service discovery on the Web might incorporate dedicated tools in the example of Hamza [4] or the Google Service Catalogue [20]. The catalogue provides an index to Google services with links to the description. The services in the catalogue are annotated and indexed using extra properties but the annotation scheme does not handle ambiguities. A more elaborate solution requires that services are first annotated and then instantiated to the ontology. Searching using SPARQL might also take advantage of reasoning for detecting inconsistencies and for discovering new inferred relations in service descriptions. In author’s thesis [7] we show how the PetStore and USPTO services [21] are annotated and instantiated to the OpenAI ontology using Algorithm 1. In this thesis, several SPARQL queries are discussed. The query of Listing 1.7 retrieves operations on pets. In this example, operations are grouped by their Tag value Pet which, in turn, is associated with Pet object using the x-on-resource property. In addition, Cat and Dog are defined as specializations of Pet object using x-kindOf property. Due to inheritance (as of Listing 1.1), the query retrieves operations on pets and on all its sub-categories (i.e. cats and dogs) rather than operations on pets alone.

Listing 1.7. An example query on Petshop.

```sparql
PREFIX openapi: <http://www.aWebsite.xxx.xxx/ns/open-api>
PREFIX po: <http://www.productontology.org/doc/>

SELECT ?operationId
WHERE {
  ?schemaInd openapi:supportedOperation ?operationInd .
  ?operationInd openapi:operationId ?operationId .
}
```

6 Conclusions

The ontology is currently being extended to support LINKS and CALLBACKS properties. Instantiating the APIs of GURU catalogue is an issue for future work. The instantiated ontology will services in the GURU catalogue will be available on the Web. At present time, a Web application which supports uploading of REST services and their instantiation to the ontology is available

20 https://www.googleapis.com/discovery/v1/apis/
21 https://github.com/OAI/OpenAPI-Specification/tree/master/examples/v3.0
22 https://apis.guru/browse-apis/
23 http://www.intelligence.tuc.gr/semantic-open-api/
on the Web for use and evaluation. Future improvements to the ontology and algorithm will incorporate comments received by its users worldwide. Query formulation on the catalogue can be facilitated by Graphical User Interfaces allowing users to select service properties from pull-down menu. Alternatively, a dedicated query language in the spirit of SOWL-QL [8] will be designed so that the user need not be familiar with the peculiarities and syntax of the underlying representation. In a more ambitious scenario, we foresee a mechanism that is capable of understanding the intention of the service being created so that the service can be implemented as a synthesis of existing services.

References