Web services are published in service registries on the Web by various software vendors to be easily discovered and re-used in applications. The Semantic Web vision provides the technology means for unifying the world of Web services and suggests representing the services as semantic objects accessible on the Web. Leveraging the latest results for hypermedia-based construction of Web APIs (i.e. Hydra) and the newest update of the OpenAPI specification, we propose a reference ontology for REST services along with a formal procedure for converting OpenAPI service descriptions to instances of this ontology. At the heart of the approach is a model for enhancing the meaning of Schema properties (i.e. re-usable JSON Schema properties commonly used to clarify the meaning of service components). Schema properties are semantically annotated (i.e. their meaning is mapped to a semantic model) or, existing properties are combined to form complex composed or polymorphic expressions. The complete algorithm for mapping service descriptions to the OpenAPI ontology is implemented and is available as a Web Application for testing.

Keywords: Web service; Ontology; OpenAPI; REST; Hydra; SHACL.

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 standardizing technologies for service publishing and discovery is of crucial importance for their adoption and market success. OpenAPI Specification is a widely adopted standard endorsed by Linux Foundation and supported by large software vendors like Google, Microsoft, IBM, Oracle, and many others. OpenAPI format is based on JSON (or YAML) and comprises a large set of properties for composing service descriptions. OpenAPI 3.0 is the first major update of the specification released in 2017. Version 3.1 (as of February 2021) provides full JSON Schema support (i.e. all keywords of JSON Schema vocabulary can be used in OpenAPI 3.1). An extension introduced by version 3.1 is that Paths are now optional (i.e. Webhooks can be used instead).

*Corresponding author
*https://www.openapis.org
The syntactic binding of OpenAPI format to JSON (or YAML) complicates the detection of similarities, inconsistencies, or of ambiguities in service descriptions. In addition, OpenAPI 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 and thus provides limited support to service discovery and composition. Therefore, in [1] we analyzed the causes of ambiguity and proposed a reference ontology for OpenAPI descriptions of REST services. This is the first work in the literature to resolve ambiguities in OpenAPI version 3.0 descriptions.

Our work was influenced by Hydra [2], a promising technology for understanding and constructing Web services that meet the HATEOAS [3] requirement of REST architectural style. Hydra provides a vocabulary for describing service capabilities that are advertised during run-time to the client in order to discover the available actions and resources. However, the Hydra vocabulary cannot fully capture all the information provided by an OpenAPI service description. Therefore, the OpenAPI v3.0 ontology was designed to incorporate the majority of Hydra ontology concepts for modeling service operations along with models not supported in Hydra (e.g. REST security features, headers, constraints). Classes together with constraints on class properties are described using SHACL [4] allowing service descriptions to be validated against the ontology. The instantiation of OpenAPI service descriptions to ontologies is outlined in [1] and [4].

The proposed ontology instantiation algorithm handles all OpenAPI objects and their properties (e.g. paths, operations, as well as re-usable objects defined under the components object such as security and schemas). Each OpenAPI document includes information spread among objects and properties in a deeply nested (i.e. tree) structure. The same property may appear many times in the same document (with the same or different meaning, or their meaning cannot be defined at all), with different scopes (local property declarations overshadow global ones) or, it can be nested inside other properties. At the heart of the proposed approach is a model for enhancing the meaning of Schema properties. Schema objects are semantically annotated (i.e. their meaning is mapped to a semantic model using keywords) or, existing properties can be combined in order to form complex composed or polymorphic expressions. The proposed algorithm handles all these issues.

The present work summarizes, extends, and integrates all previous work by the authors [1,4]. This includes (a) an OpenAPI annotation mechanism that eliminates ambiguities in service descriptions, (b) an ontology for REST services, and (c) exploitation of SHACL language for validating Schema descriptions against the ontology. It extends the ontology instantiation mechanism for manipulating all OpenAPI elements in service descriptions. The emphasis is on Schema objects. The system is implemented in a Web application [5] that supports conversion of OpenAPI service

---

[1] https://restfulapi.net/hateoas/
descriptions to instances of OpenAPI ontology, downloading the resulting ontology and, searching for web services that meet certain criteria using SPARQL. For a complete discussion, the reader is referred to [5].

OpenAPI and OpenAPI ontology are complementary representations and do not replace one another. OpenAPI services are published in Web service repositories by their software vendors. The advantages of using OpenAPI are many: it is a complete framework that is supported by a large set of tools Swagger editor, code generation for multiple languages, and frameworks like Javascript, Node.js, User Interfaces, etc.). On the other hand, the motivation for using ontologies is that they are closer to the way machines analyze and comprehend OpenAPI’s inherent content. It is, therefore, easier for a machine to discover similarities (e.g. using ontology query languages such as SPARQL) or, detect services with inconsistencies (e.g. using standard ontology reasoners) using ontologies. Enabling automatic service synthesis and orchestration is a long and more ambitious goal of this approach. The ontology should not be necessarily visible to the end user or API developer. It settles in the background and its role is twofold: (a) informs the user of 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. [6] for discovering OpenAPI compliant REST APIs).

The idea of using ontologies is not new. Existing ontologies fit well the needs of remote procedure call technologies such as SOAP [7]. However, the emergence of REST [8] created new difficulties for the representation of hypermedia-driven APIs (such as REST) that call for the dynamic discovery of resources at run-time (referred to as HATEOAS [9]). On the other hand, service descriptions based on ontologies are not commercialized yet or applied in real-world or large-scale service repositories. The reasons are many: one is the fragmentation of technologies spanning different disciplines from Software Engineering to Semantic Web; another is that ontologies are not yet widely accepted by the industry due to their complexity and poor run-time performance. In particular, ontologies have not been proven to be scalable for larger service repositories. Annotation and instantiation of services to ontologies is not a real-time process. The size of REST descriptions (i.e. reaching some thousands of lines in many cases) and the speed of the service instantiation to an ontology or, the search in a triple-store database, are traded-off.

Related work on services description languages (with emphasis on REST services) is presented in Sec. 2. OpenAPI specification along with the proposed annotation method for resolving ambiguities in service descriptions is discussed in Sec. 3. The OpenAPI ontology and the instantiation of service descriptions to the ontology are discussed in Sec. 4. Ontology evaluation is described in Sec. 5 followed by conclusions and issues for future work in Sec. 6.

---

[9]https://restfulapi.net/hateoas/
2. Related Work

OpenAPI Specification\(^4\) formerly known as Swagger, is a fairly new technology and despite its high impact, it has not been explored in depth in the literature. As a result, the idea of mapping OpenAPI services to an ontology is also novel and the related work is very limited.

OpenAPI is an open-source, language-agnostic specification, through which a consumer can understand and use a service by applying minimal implementation logic. Service descriptions are offered in either JSON or YAML format, which can be produced statically, or be generated dynamically from the application. This allows the design and implementation of APIs to follow either a top-down (the service description is initially created and then the service is implemented) or a bottom-up approach (the service description is generated from the service implementation). The initiative is supported by a constantly increasing community including companies like Google, Microsoft, IBM, and many others.

Syntactic description languages describe the requirements for establishing a connection with a service and the message formats to successfully communicate with it. WSDL\(^9\) for SOAP services, WSDL\(^9\) is an XML-based interface description language for SOAP services\(^7\) (i.e. how and where their functionality can be invoked). The latest version 2.0 introduced changes in the document structure as well as HTTP 1.1 protocol support in order to allow the description of REST services. Its previous version is still preferred for the description of SOAP services. SAWSDL\(^10\) defined extension properties in order to add semantics to WSDL components. SAWSDL is criticized as it comes without any formal semantics. This hinders logic-based discovery and composition of Web services and calls for software outside the framework to resolve the semantic heterogeneities\(^11\). WADL\(^12\) is yet another XML-based interface description language designed to describe REST services. It is closely related to WSDL, it attempts to model the resources provided by the service and their relationships but has limited support for describing the meaning of service resources. RAML\(^f\) and API Blueprint\(^g\) for REST are popular mainly due to their simplicity and compatibility with common machine-readable formats like XML and JSON.

Semantic approaches describe services by means of semantic models (i.e. ontologies) and are more capable of supporting automated service discovery and composition. WSMO\(^13\) defines a conceptual model and WSML language for the semantic description of Web services. OWL-S\(^14\) is an upper ontology for Web services but, similar to all other methods, does not support the dynamic discovery of resources at runtime. SA-REST\(^15\) offers a set of properties for annotating service descriptions written in HTML. The idea is similar to SAWSDL, attempting to make HTML service descriptions machine-understandable. Hydra\(^2\) simplifies the construction of

\(^4\)https://www.openapis.org
\(^5\)https://raml.org
\(^6\)https://apiblueprint.org
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 for understanding and constructing Web services that meet the HATEOAS requirement of REST architectural style.

Musyaffa et al. introduce annotations in Schema and Parameter objects for OpenAPI v2.0. They do not handle all causes of ambiguity nor do they handle OpenAPI v3.0 descriptions. The annotations appear within text properties and cannot be interpreted by a machine without pre-processing. Schwichtenberg, Gerth, and Engels map OpenAPI v2.0 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 OpenAPI v2.0 Schema objects to OWL-S using heuristics and name similarity matching techniques. The mapping is error-prone and needs to be adjusted manually. Most importantly, their choice of OWL-S model for representing REST services is controversial: OWL-S is good for SOAP services but not good for hypermedia-driven Web services like REST. Finally, they do not handle OpenAPI v3.0 descriptions. In a recent contribution, Hamza et al. propose an example-driven approach and a representation of REST APIs for discovering OpenAPI-compliant REST Web APIs. This facilitates 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. OpenAPI

OpenAPI 3.0 service descriptions comprise many objects. Figure illustrates the structure of an OpenAPI service description. Each object has a list of properties that can be objects as well. Objects and properties defined under the Components unit of an OpenAPI document can be re-used by other objects or they can be linked to each other (e.g. using keyword $ref). However, these links are not always explicitly expressed (e.g. there are properties with the same name with no reference to one another or to an external model).

The Info object provides non-functional information such as the name of the service, service provider, and terms of use. The Servers object provides information on where the API servers are located (i.e. multiple servers can be defined). The Security object contains the security schemes that the service uses for authentication (i.e. API keys, OAuth2.0 common flows and OpenID Connect). The Paths object

---

^https://blog.readme.io/an-example-filled-guide-to-swagger-3-2/
^https://oauth.net/2/
^https://auth0.com/docs/protocols/openid-connect-protocol
contains the relative paths of the service endpoints. Each Path item describes the available operations based on HTTP methods (e.g. get, put, post).

The core of an OpenAPI document is the Operation object that provides the needed information for expressing HTTP requests to the service. It provides information regarding the HTTP responses of the service. The Components object holds a set of reusable objects. These objects can be responses, parameters, schemas, request bodies, and more. The Responses object specifies the expected responses of an Operation and maps each response to an HTTP status code and to any HTTP Headers that an operation’s response may return. The Parameters object specifies the parameters that operations can use. These can be path parameters (i.e. specified in the operation’s path), query parameters (which are appended to the URL when sending a request), header, or cookie parameters.

The Schemas object specifies the data type that can be used to describe the request and response messages. It can be a primitive (string, integer), an array, or a model. For the definition of Schema objects, the specification resorts to JSON Schema. A Schema object definition can be enhanced with XML data types. Finally, new data types can be defined as a combination of a specification of existing ones (i.e. using the allOf property or, the oneOf property, respectively).

3.1. Semantically Annotated OpenAPI

Despite its rigorous service language format (JSON or YAML), OpenAPI service descriptions can be vague. There can be properties that share the same meaning (although they are defined using different names) or, their meaning is ambiguous.
Probably, a human can easily resolve ambiguities either by the element names or by the description that may be provided with the properties. However, in order for a machine to act similarly to a human it is necessary to provide additional information, that clarifies ambiguous properties in an OpenAPI service description. In recent conference papers \(^1\) we analyzed the causes of ambiguity. In order to eliminate ambiguities, OpenAPI properties must be semantically annotated and associated with entities of a semantic model (e.g. using Schema.org vocabulary). OpenAPI descriptions are annotated using the extension properties defined in Table \(^2\).

The \(x\)-\textit{refersTo} extension property specifies the association between an OpenAPI element and a concept in a semantic model. The \(x\)-\textit{refersTo} extension property in Listing \(^3\) is used to semantically annotate a Pet model and its properties: it associates the model with Pet class in Product ontology\(^4\) Listing \(^3\) demonstrates how polymorphism is expressed using the discriminator property: the model of a Pet uses the \textit{name} property as a discriminator in order to determine whether this is a model of a dog or of a cat. In the same example, the \textit{allOf} property is used to specify that the model of a dog or a cat model has the properties of a Pet model and also additional properties. In the same example, if a model describes a specific group of pets (e.g. dogs), the \(x\)-\textit{kindOf} extension property is used instead to denote that the model is a subclass of the referred semantic concept.

While \(x\)-\textit{refersTo} and \(x\)-\textit{kindOf} link OpenAPI elements with concepts from a semantic model, the \(x\)-\textit{mapsTo} is used to semantically associate an OpenAPI element with another OpenAPI element within the same OpenAPI document. The OpenAPI element using the \(x\)-\textit{mapsTo} property is considered semantically similar to the OpenAPI element it refers to. In Listing \(^3\) the \(x\)-\textit{mapsTo} property is used in Parameters object to dictate that query parameter \textit{name} refers to the property

\(^{1}\)http://www.productontology.org/
name of Schema Object Pet. There is a semantic similarity between the id property of Pet model and the petId property in https://schema.org/identifier which later appears in the document as petId (not shown in the example). Similarly, Schema Object Pet should refer to http://www.productontology.org/doc/Pet and property name refers to http://purl.org/goodrelations/v1#name.

Listing 1. OpenAPI model polymorphism example.

```json
parameters:
  Query:
    name: name
    in: query
    required: true
    schema:
    type: string
    x-mapsTo: '#/components/schemas/Pet.name'
schemas:
  Pet: # A Pet model extended with Annoated OpenAPI v3.0 properties
  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/identifier
  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 is described using the array type. However, it is very common for a collection’s definition to be 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 2 defines a model as a collection of Pet objects (totalItems property denotes population).


```json
schemas:
```
Annotated OpenAPI Descriptions and Ontology for REST Services

PetCollection: # A Pet Collection definition

type: object

properties:
  pets:
    type: array
    items:
      $ref: '#/components/schemas/Pet'
    totalItems: type: integer

The x-onResource extension property is used in Tag Objects to specify the resource that a tag refers to. Tags are used to group operations either by resources or 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 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 3, the 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 3. Associating Schema objects with a Tag.

```
tag:
  name: pet
  description: Everything about your Pets
  externalDocs:
    description: Find out more
    url: 'http://swagger.io'
  x-onResource: '#/components/schemas/Pet'
paths:
  /pet/findByStatus:
    get:
      x-operationType: 'http://schema.org/SearchAction'
      tags:
        - pet
      summary: Finds Pets by status
      description: Multiple status values can be provided
      operationId: findPetsByStatus
      parameters:
        - $ref: '#/components/parameters/statusQuery'
      responses:
        200:
          description: successful operation
          content:
            application/json:
      schema: $ref: "#/components/schemas/Pets"
        400:
          description: Invalid status value
          security:
            - petstore_auth:
              - 'write:pets'
              - 'read:pets'
```

Finally, the x-operationType extension property is used to specify the type of an Operation object. A request is characterized by an HTTP method but, the semantics of the HTTP methods are too generic and may have a more specific meaning. In Listing 4, the x-operationType extension 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
Mainas, Boura mis, Karavasil eiou, Pet rakis

semantically describes the operation type. The Action\(^\text{m}\) type of the Schema.org vocabulary provides a detailed hierarchy of Action sub-types that can be used by the property.

Listing 4. Clarifying the meaning of an operation.

```json
paths:
/pet/findByStatus:
get:
x-operationType: 'http://schema.org/SearchAction'
tags:
- pet
summary: Finds Pets by status
description: Multiple status values can be provided with comma separated strings
operationId: findPetsByStatus
produces:
- application/xml
- application/json
parameters:
- 
$ref: '#/parameters/statusQuery'
responses:
'200':
description: successful operation
schema:
$ref: '#/definitions/PetCollection'
'400':
description: Invalid status value
security:
- petstore_auth:
  - 'write: pets'
  - 'read: pets'
```

4. OpenAPI Ontology

The semantic meaning of the service as a whole is captured by the OpenAPI ontology of Figure 4. At the heart of the ontology is Hydra core vocabulary. Schemas, Hydra is enhanced with additional semantic models representing information for security, headers, and constraints.

Document class represents the documentation and the entry point of the service. Similarly to the OAS structure, it provides general information (Info class) regarding the described service and also specifies the service paths and the entities that it supports. The Path class represents (relative) service paths (through the pathName property). The Operation class provides information for sending HTTP requests. Request bodies are represented by the Body class, while, responses are declared in the Response class specifying the status code and the data returned. 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. OpenAPI parameters are separate classes for every parameter type. The Header class provides all the definitions of header parameters. The Cookie class defines the cookies that are sent with HTTP requests. The Parameter class defines all parameters that are attached to the operations’ URL. The class

\(^{m}\text{http://schema.org/Action}\)
Fig. 2. OpenAPI 3.0 ontology.

Annotated OpenAPI Descriptions and Ontology for REST Services

has PathParameter and Query sub-classes that refer to the corresponding path and query parameters of the specification. Request and response bodies, 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). Figure 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.

Table 2 provides some insights into the OpenAPI ontology. Sub-classes are mainly due to the various supported Security schemes, the OAuth2 flows and the organization of HTTP responses based on the HTTP status code family (i.e. 1xx, 2xx, 3xx, 4xx, 5xx). The number of Individuals occurred from the definition of supported HTTP Methods, as well as the different styles that can be applied to the formatting of data in Parameter Objects. Ontology metrics do not consider
the SHACL vocabulary that is used for the description of Schema Objects in the
OpenAPI ontology.

Table 2. OpenAPI ontology metrics

<table>
<thead>
<tr>
<th>Classes</th>
<th>Subclasses</th>
<th>Object Properties</th>
<th>Data Properties</th>
<th>Individuals</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>17</td>
<td>34</td>
<td>45</td>
<td>15</td>
</tr>
</tbody>
</table>

4.1. Instantiating OpenAPI Service Descriptions to the Ontology

Algorithm 1 summarizes the ontology instantiation process. The algorithm scans
the OpenAPI document and instantiates OpenAPI objects to ontology classes. In
particular, after uploading the ontology in the memory, the algorithm will scan the
OpenAPI file to extract info, servers, security schemes, tags, and paths objects.
These objects will become individuals of their corresponding classes.

Initially, the ontology model is initialized and an individual of class Document is
created (OpenAPI object is mapped to class Document). Function `parseInfoObject`
creates an instance of the Info class from information defined in Contact and License
objects which, in turn, are mapped to individuals of their corresponding classes.
Listing 5 illustrates the mapping of the Info object to the ontology.

The next step is the conversion of Server Objects into the respective individuals
of the Server class (lines 5-8). There may exist more than one occurrence of servers
in an OpenAPI file, therefore they are stored in a list for later use (line 5). The
list will be overwritten by server information defined in Path or Operation objects.
Security Schemes are converted into individuals of the respective Security classes
(Figure 3) through method `parseSecuritySchemeObject`, described in Algorithm 2.
A Security scheme object becomes an individual of a certain class (ApiKEY, HTTP,
Algorithm 1 Converting OpenAPI object to ontology

1: procedure parseDocumentObject(OpenAPI document)
2: 
3: Init Ontology Model, docInd ← create Document individual
4: infoInd = parseInfoObject(Info Object) ▷ Info individual
5: docInd add relation openapi:info with infoInd
6: globalServerInds ← init List
7: for Server Object in servers property do ▷ from OpenAPI Object
8: serverInd = parseServerObject(Server Object) ▷ Server individual
9: globalServerInds add serverInd
10: for Security Scheme Object in securitySchemes property do ▷ from Components Object
11: parseSecuritySchemeObject(Document individual, Security Scheme Object)
12: globalSecReqObjects ← init List
13: for Security Requirement Object in security property do ▷ from OpenAPI Object
15: globalSecReqObjects add secReqInd
16: tagShapeMap ← init Map ▷ pairs of Tags and Shapes individuals
17: schemaObjects ← schemas property ▷ from Components Object
18: for Tag Object in tags property do ▷ from OpenAPI Object
19: Pair(Tag, Shape) = parseTagObject(Tag Object, schemaObjects)
20: tagShapeMap add Pair
21: for Path Object in Paths Object do
22: pathInd ← Create Path individual
23: pathServerInds ← init List
24: for Server Object in servers property do ▷ from Path Item Object
25: serverInd = parseServerObject(Server Object) ▷ Server individual
26: pathServerInds add serverInd
27: pathParameterObjects ← parameters property ▷ from Path Object
28: Select serverIndList (pathServerInds if not empty, otherwise globalServerInds)
29: for Operation Object in Path Item Object do
30: parseOperationObject(docInd, pathInd, Operation Object, tagShapeMap, pathParameterObjectss, serverIndList, globalSecReqObjects)

OpenIDConnect, or OAuth2) depending on the property type specified inside the object. The security individual is then connected to the Document individual with property supportedSecurity. Handling Security Requirement objects (lines 11-14) is a case similar to that of Server Objects. Security Requirement objects are converted
to the respective individuals and are kept in a list, which may be overwritten by Security Requirements defined in Operation objects.

Listing 5. Instantiating an OpenAPI Document object.

```plaintext
ex: PetStoreDocument
  a openapi:Document ;
openapi:info { 
  openapi:serviceTitle "Sample Pet Store App" ;
  openapi:description "This is a sample server for a pet store." ;
  openapi:termsOfService <http://example.com/terms/> ;
openapi:contact { 
  openapi:creator "API Support" ;
  openapi:url <http://www.example.com/support> ;
openapi:email <support@example.com> .
} ;
openapi:license { 
  openapi:licenseName "Apache 2.0" ;
  openapi:url <http://www.apache.org/licenses/LICENSE-2.0.html> 
} ;
... 
```

Algorithm 2  Create individuals of Security class.

```plaintext
function PARSESECURITYSCHEMEOBJECT(Individual document, Security Scheme object)
  type ← object get type
  if type is apiKey then
    secInd ← create individual of ApiKey Class
  else if type is http then
    secInd ← create individual of Http Class
  else if type is oauth2 then
    secInd ← create individual of OAuth2 Class
  else if type is openIdConnect then
    secInd ← create individual of OpenIDConnect Class
  document add relation openapi:supportedSecurity with secInd
```

The conversion algorithm [1] resumes by handling Tag objects (lines 15-19). The x-onResource property associates a Tag Object with a Schema object. To track this relation between the objects, a map is used containing pairs of Tags and Schemas. A pair is generated from the function parseTagObject (Algorithm 3), which creates a Tag individual along with all its properties and also handles the use of x-onResource property. If the property exists, it creates a Shape individual from the referred Schema object and returns the pair of Tag-Shape individual, otherwise, it returns only the Tag individual. The pair is added to the appropriate map that will be used during the instantiation of Operation objects.

Listing 6. Instantiating Path and Operation objects to the ontology.
Algorithm 3 Create individuals of \texttt{Tag} class.

1: function \texttt{parseTagObject}(Tag object, schemaList) ← schemaList, List of Schema objects

2: \hspace{1em} tagInd ← Create Tag individual

3: \hspace{1em} if Tag object contains property x-onResource then

4: \hspace{2em} schema ← from schemaList \hspace{1em} ▷ find using value of x-onResource

5: \hspace{2em} shape = parseSchemaObject(schema name, schema, schemaList) \hspace{1em} ▷ Shape individual

6: \hspace{2em} return Pair of tagInd and shape

7: \hspace{1em} else

8: \hspace{2em} return Pair of tagInd and NO Shape

The last step of the conversion algorithm is the handling of Path objects (lines 20-29). For every Path object, a path individual is created and the function \texttt{parseServerObject} instantiates any servers defined in the Path object. Server individuals are kept in a list that represents the servers used in the current path. If the property servers is not set, then the list is replaced by the globalServerList defined earlier. Then, the algorithm extracts all the parameters defined in the current path that will be used on the instantiation of Operation objects (line 29). An Operation Object may overwrite a lot of the previously mapped OpenAPI objects such as servers, parameters, and security requirements. For this reason, these OpenAPI ob-
jects are given as arguments to the function parseOperationObject. Listing 5 shows the individuals of Path and Operation objects of Listing 3 to the ontology.

Function parseOperationObject (algorithm 4) is responsible for the instantiation of Operation objects. Initially, the algorithm checks if the x-operationType property exists in the objects and uses its value to make the operation individual also an individual of the referred class. Regarding tags, the algorithm uses the tagShapeMap to retrieve the respective tag individuals and assign them to the operation individual. Additionally, if the retrieved tag is linked with a Shape (Schema object) then the operation individual is added as a supported operation in the Shape individual. For security requirements and servers, the algorithm checks the respective properties of the operation object and if they don’t have values, then it uses the server and security requirement list from the function’s input (lines 13-28).

Operation’s parameters are defined through the function parseParameterObject (algorithm 5), which is responsible to instantiate them to the corresponding classes based on their type (path, cookie, header, query). Functions parseQueryParameterObject, parsePathParameterObject, parseCookieParameterObject, and parseHeaderParameterObject share the same logic. In all functions, the respective individual is created, and a Schema object is extracted that will be converted to Shape through the function parseSchemaObject.

The last step of the function parseOperationObject is the instantiation of the Request body and Response objects. Function parseRequestBodyObject is responsible for handling Request Body objects. After creating the Body individual, the function collects all Media Type objects and for each Media Type calls function parseMediaTypeObject to create the individual and then associate it with the Body individual. Depending on the value statusCode of the Response Object, function parseResponseObject creates an individual of the corresponding class. In order to map the properties of the Response object two lists are created, one for the Header individuals and one for the Media Type individuals which are handled by functions parseHeaderObject and parseMediaTypeObject respectively. The list of Header individuals is mapped to responseHeader property of the Response Individual. The list of Media Type Individuals is mapped to the content property of the Response individual.

4.2. Instantiating Schema Objects

Schema objects are mainly defined under the Components section of an OpenAPI document and can be referenced by every other object. They are of high significance due to their frequent appearance in core elements of OpenAPI descriptions. Schema objects are mapped to classes using SHACL. SHACL can be used to describe and validate the structure of RDF data, similar to XML-Schema or JSON Schema. It provides an RDF vocabulary of built-in types of constraints (e.g. cardinality: minCount, maxCount). Table 3 shows how the allOf, oneOf, anyOf and not JSON Schema properties are expressed in SHACL. This allows for expressing Schema objects and their classes as compositions of existing ones. Polymorphism allows
Algorithm 4 Create individual of Operation class.

1: function parseOperationObject(docInd, pathInd, Operation object, tagShapeMap, pathParameterObjects, serverInds, globalSecReqInds, schemaList)
2:     opInd ← Create Operation individual
3:     if Operation's property x-operationType has value then
4:         make opInd also individual of x-operationType referred class
5:     opInd add relation openapi:onPath with Path individual
6:     opInd add relation openapi:supportedOperation with Document individual
7:     for tag in tags property do ▷ from Operation object
8:         if tagShapeMap contains tag then
9:             pair ← from tagShapeMap
10:            opInd add relation openapi:tag with tagInd ▷ from pair
11:            if pair has Shape then
12:                Shape add relation openapi:supportedOperation with opInd
13:     securityReqObjects ← security property ▷ from Operation object
14:     if securityReqObjects is empty then
15:         for secReqInd in globalSecurityReqIndividuals do
16:             opInd add relation openapi:security with secReqInd
17:     else
18:         for Security Requirement object in securityReqObjects do
19:             secReqInd = parseSecurityReqObject(Security Requirement object)
20:             opInd add relation openapi:security with secReqInd
21:     serverObjects ← servers property ▷ from Operation object
22:     if serverObjects is empty then
23:         for servInd in serverInds do
24:             opInd add relation openapi:serverInfo with servInd
25:     else
26:         for Server object in serverObjects do
27:             servInd = parseServerObject(Server object) ▷ Server individual
28:             opInd add relation openapi:serverInfo with servInd
29:     parameterObjects ← parameters property ▷ from Operation object
30:     merge parameterObjects with pathParameterObjects
31:     for parameter in parameterObjects do
32:         parseParameterObject(parameter, opInd, schemaList)
33:         reqBodyInd = parseRequestBodyObject(Request Body Object, schemaList) ▷ Body individual
34:             opInd add relation openapi:requestBody with reqBodyInd
35:     for Response Object in responses property do ▷ from Operation Object
36:         respInd = parseResponseObject(statusCode, response Object, schemaList) ▷ Response individual
37:             opInd add relation openapi:response with respInd

Annotated OpenAPI Descriptions and Ontology for REST Services
Algorithm 5 Handling of Parameter Object

1: function parseParameterObject(Parameter object, Operation individual, schemaList )
   → schemaList, List of Schema objects
2:    type ← parameter get property in
3:   if type is query then
4:      queryInd = parseQueryParameterObject(parameter, schemaList) → QueryParam individual
5:      individual add relation openapi:parameter with queryInd
6:   else if type is path then
7:      pathParamInd = parsePathParameterObject(parameter, schemaList) → PathParam individual
8:      opInd add relation openapi:parameter with pathParamInd
9:   else if type is cookie then
10:     cookieInd = parseCookieParameterObject(parameter, schemaList) → Cookie individual
11:     opInd add relation openapi:cookie with cookieInd
12:   else if type is header then
13:     headerInd = parseHeaderParameterObject(parameter name, parameter, schemaList) → Header individual
14:     opInd add relation openapi:requestHeader with headerInd

Algorithm 6 Handling of Request Body and Response Objects

1: function parseRequestBodyObject(Request Body object, schemaList )
2:    reqBodyInd ← create Body individual
3:    mediaTypeList ← from property content
4:   for mediaType in mediaTypeList do
5:      mediaTypeInd = parseMediaTypeObject(mediaType name, mediaType, schemaList)
6:      reqBodyInd add relation openapi:content with mediaTypeInd
7:    return reqBodyInd

1: function parseResponseObject(statusCode, Response object, schemaList)
2:    respInd ← create Response individual → based on status code family
3:    respHeaders ← from property headers
4:   for headerObj in respHeaders do
5:      headerInd = parseHeaderParameterObject(header name, headerObj, schemaList)
6:      respInd add relation openapi:responseHeader with headerInd
7:    respMediaTypes ← from property content
8:   for mediaType in respMediaTypes do
9:      mediaTypeInd = parseMediaTypeObject(mediaType name, mediaType, schemaList)
10:     respInd add relation openapi:content with mediaTypeInd
11:    return respInd
Algorithm 7 Handling of Schema Objects

1: function parseSchemaObject(schema name, Schema object, schemaList)
2: if Schema object has one of the properties allOf, oneOf, anyOf then
3:     shape = createNodeShape(schema name, Schema object, schemaList)
4:     innerSchemas ← get Schema Objects from properties allOf, anyOf, oneOf
5:     for schema in innerSchemas do
6:         innerShape = parseSchemaObject(schema name, schema, schemaList)
7:         shape add relation with innerShape ▷ based on mapping of Table 3
8:     return shape
9: else
10:     schemaType ← from property type
11:     if schemaType = object then
12:         shape = createNodeShape(schema name, Schema object, schemaList)
13:         return shape
14:     else if schemaType = int or boolean or string then
15:         shape = createPropertyShape(empty, schema name, schema object, schemaList)
16:         return shape
17:     else if schemaType = array then
18:         shape = createCollectionNodeShape(schema name, Schema object, schemaList)
19:         return shape

different views on objects (i.e. a Schema object that accepts alternative datatypes).

Table 3. Mapping Schema properties to SHACL.

<table>
<thead>
<tr>
<th>Schema Object property</th>
<th>SHACL property</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

Each Schema object is mapped to a Shape class (shown in ovals in Figure 2) in the ontology using SHACL. The Shape Class is distinguished into NodeShape class and PropertyShape class. The NodeShape class represents the classes that describe the models of an OpenAPI service description (their Schemas) and, PropertyShape class represents the properties of a class, their datatype(s), and restrictions. A Shape class determines how to validate a focus node (a node from the data graph) based on the values of properties and other characteristics of the focus node. The two types of Shape classes are defined in SHACL. A NodeShape contains targets that specify which nodes in the data graph must conform to a Shape and constraint components.
that determine how to validate a node. Except for \texttt{sh:targetNode}, which specifies
directly the nodes to be validated, there is also a \texttt{sh:targetClass} to denote that all
the nodes of a given type need to conform with a particular Shape. Finally, \texttt{sh:path}
of a PropertyShape points at the URI of the property that is being restricted.

Algorithm 7 provides an overview of the process to convert Schema Objects
into Shapes. The algorithm based on the Schema object’s type performs the appro-
priate conversions to generate the respective Node and Property Shapes. Function
\texttt{createNodeShape} generates the NodeShape from the given Schema Object and also
checks for any usage of extension properties and treats them respectively. Then, for
every property of the Schema Object function \texttt{createPropertyShape} is called that
will create the respective PropertyShape, which will be linked to the NodeShape
through property \texttt{sh:property}. Function \texttt{createPropertyShape} is responsible for defining
PropertyShape individuals. At first, the function will define the PropertyShape
and then it will for any usage of extension properties, and perform the respective
actions for each case in order to associate a semantic concept to the Shape. Func-
tion \texttt{createCollectionNodeShape} handles Schema Objects of type array. The function
generates a NodeShape associated semantically with a subclass of the OpenAPI on-
tology Collection class. Then, it calls function \texttt{createPropertyShape} for the Schema
Objects defined under the item property and maps them with the created Node-
Shape individual. The detailed process of converting Schema Objects into Shapes
can be found in \[5\] and \[4\].

Listing 7. Pet object example.
\begin{verbatim}
1 | Pet:  
2 |   type: object  
3 |   required:  
4 |     - id  
5 |   properties:  
6 |     id:  
7 |       type: integer  
8 |       format: int64
\end{verbatim}

Listing 7 shows the model of a Pet with property \texttt{id}. Listing 8 is the correspond-
ing Shape class. The model becomes an instance of the NodeShape class with the
target class Pet. The \texttt{rdfs:label} predicate is human-readable information. The Shape
node has one property (Pet\_idPropertyShape in this case) of type PropertyShape
of type Pet\_id which is the class that is being restricted.

Listing 8. Shape and Property Shape nodes of Pet object.
\begin{verbatim}
1 | <PetNodeShape> a sh:NodeShape ;  
2 |   rdfs:label "PetNodeShape" ;  
3 |   sh:property <Pet_idPropertyShape> ;  
4 |   sh:targetClass <Pet> .  
5 | <Pet> a owl:Class .  
6 | <Pet_idPropertyShape> a sh:PropertyShape ;  
7 |   rdfs:label "Pet_idPropertyShape" ;  
8 |   openapi:name "id" ;  
9 |   sh:datatype xsd:long ;
\end{verbatim}
Listing [8] shows that each \textit{sh:property} becomes an instance of \texttt{PropertyShape} class. The values of \textit{sh:datatype} come from the type and format of each property (i.e. \texttt{int32} has value \texttt{xsd:int} and \texttt{int64} have value \texttt{xsd:long}). The \textit{sh:path} predicate points at the URI of the property that is being restricted.


```
Pet:  
ex-refersTo: https://example.com/ontology/Pet  
type: object  
required:  
- id  
properties:  
- id:  
ex-refersTo: https://example.com/ontology/Id  
type: integer  
format: int64
```

Listing 10. Shape models with \textit{x-refersTo} properties.

```
<PetNodeShape> a sh:NodeShape ;  
rdfs:label "PetNodeShape" ;  
sh:property <Pet_idPropertyShape> ;  
<Pet_idPropertyShape> a sh:PropertyShape ;  
rdfs:label "Pet_idPropertyShape" ;  
openapi:name "id" ;  
sh:datatype xsd:long ;  
sh:path <https://example.com/ontology/Id>. 
```

In Listing [9] both, the Pet Schema object and its property are semantically annotated. In particular, \textit{x-refersTo} associates the object and its property to a semantic value (class) which becomes the target and path objects of their corresponding Shapes. This is shown in Listing [10]. If the extension property is \textit{x-kindOf} the target and path objects become sub-classes of the designated models. Listing [10] must be modified as shown in Listing [11]. The \textit{x-mapsTo} annotation property can also be used in a Schema object to designate that it points to another Schema object or property Schema with the same semantics. It is handled similarly to \textit{x-refersTo} but the target and path objects now link to the model that the \textit{x-mapsTo} property refers to.

Listing 11. Shape models with \textit{x-kindOf} properties.

```
<PetNodeShape> a sh:NodeShape ;  
rdfs:label "PetNodeShape" ;  
sh:property <Pet_idPropertyShape> ;  
<Pet> a owl:Class ;  
```
There are cases where a Schema Object should not have a semantic value: an OpenAPI description contains Schemas that are not widely used or, they are so uniquely written that do not contribute to a general purpose. For example, an author of an API creates a Response object for the debugging of the server or some other very specific purpose. This Response Object is of no use to other authors of OpenAPI descriptions and therefore, it is not necessary to acquire a semantic value. A way around this is to write x-refersTo: none. The corresponding node Shape will not have a sh:targetClass predicate or, if it is a property Shape, it will not have a sh:path predicate. Listing 12 shows the resulting Node and Property Shapes.

Listing 12. Shape models when x-refersTo takes value none.

The x-collectionOn property is used to indicate that a Schema Object is actually a collection. This is showcased in Listing 13. Collections are represented using openapi:Collection class. The target class is PetCollection and will become a subclass of the general Collection class. The Shape Property class corresponds to property pets and declares that each member of the collection is a pet. This is showcased in Listing 14.


Composition: Schema objects can also be defined using any of the keywords `allOf`, `anyOf`, `oneOf`, and `not`. The `allOf` is used to define new Schemas as a composition of the Schemas referred to after the keyword. In fact, the `allOf` keyword expresses the concept of inheritance: the resulting composite schema inherits the properties of all referred Schemas. Keywords `oneOf` and `anyOf` are used to express the concept of polymorphism: they are used to define new Schemas that can take the form of one or more alternative Schemas. Finally, keyword `not` is used to restrict a Schema (i.e. it declares which type of value is not accepted as value for an object property).

The Pet Schema object of Listing 15 extends the OldPet with the property `id`. The PetNodeShape is defined using SHACL predicate `sh:and` in order to express that it accepts the properties of the focus node OldPetNodeShape (i.e. the Shape of the node being extended) and the additional Shape property of a blank node. The new class must inherit the properties of the original class. The way to express this is to define PetNodeShape as a sub-class of OldPetNodeShape class.

Listing 15. Object composition with `allOf`.

```json
Pet:
  allOf:
  - $ref: '#/components/schemas/OldPet',
  - type: object
    required:
    - id
    properties:
      id:
        type: integer
        format: int64
OldPet:
  type: object
  required:
  - name
  - tag
  properties:
    name:
      type: string
    tag:
      type: string
```

Listing 16. Composed Shape models.

```json
<PetNodeShape> a sh:NodeShape;
sh:and ( [ a sh:NodeShape;
  sh:property <Pet_idPropertyShape>
] )
</OldPetNodeShape>
```
The addition of semantic annotations in composed Schema Objects is translated to inheritance relations between semantic models. For example, the extension property \textit{x-refersTo: https://example.com/ontology/Pet} is added in Listing 15 before the \textit{allOf} keyword. The algorithm will create a class with the name of the model referred to. This will become the target class of the new model (i.e. \textit{sh:targetClass <https://example.com/ontology/Pet>} will replace line 8 in Listing 16). Besides, the OldPet Schema can be annotated as well by adding an extension property \textit{x-refersTo: https://example.com/ontology/OldPet} (i.e. between lines 12 and 13 in Listing 15). Then, the target class of the object being extended will be the class referred to in the extension property (i.e. line \textit{sh:targetClass <https://example.com/ontology/OldPet>} will replace line 13 in Listing 16). Also, class \textit{<OldPet>} in line 20 will not be created. In line 18, the model \textit{http://example.com/ontology/OldPet} will become a superclass. Listing 17 shows the resulting composed Shape class when both models are annotated. There is also a case where either annotation property (i.e. for Pet or OldPet) has value \textit{none}. This has the following impact on the instantiation process: (a) no target class will be created for any model annotated with \textit{none}, (b) the Shape models for both, Pet and OldPet will be created but, (c) there will be no sub-class relationship between them.

Listing 17. Annotated composed Shape models.

```xml
<sh:targetClass <Pet>
<OldPetNodeShape> a sh:NodeShape ;
rdfs:label "OldPetNodeShape" ;
sh:property <OldPet_tagPropertyShape>, <OldPet_namePropertyShape> ;
sh:targetClass <OldPet>
<Pet> a owl:Class ;
rdfs:subClassOf <OldPet> .
<OldPet> a owl:Class .
```
Polymorphism: Often API requests and responses are described by alternative Schemas. In OpenAPI this can be expressed using keywords oneOf and anyOf. The Shape model is expressed using SHACL predicates sh:xone and sh:or respectively. Listing [18] shows an unnamed Schema for response with code "201" which can be one of Dog or Cat. Because the Schema has no name, the resulting Shape node of Listing [19] is a blank NodeShape (i.e. a Shape without a URI) with predicate sh:xone and an RDF list as object and member individuals CatNodeShape and DogNodeShape. The case with keyword anyOf is handled likewise; the only difference is that predicate sh:xone in line 3 of Listing [19] will be replaced by keyword sh:or.

Listing 18. Polymorphed un-named object.
```
1 responses:
2  "201":
3   description: variety
4   content: application/json:
5     schema:
6       oneOf:
7         - $ref: "/components/schemas/Cat"
8         - $ref: "#/components/schemas/Dog"
```

Listing 19. Polymorphed Shape object.
```
1 openapi: schema [ a sh:NodeShape ;
2     sh:xone ( <DogNodeShape> <CatNodeShape> ) ]
3
4 <CatNodeShape> a sh:NodeShape ;
5   rdfs:label "CatNodeShape" ;
6   sh:property <Cat_agePropertyShape>,<Cat_huntsPropertyShape>;
7   sh:targetClass <Cat> .
8
9 <Cat> a owl:Class .
10
11 <DogNodeShape> a sh:NodeShape ;
12   rdfs:label "DogNodeShape" ;
13   sh:property <Dog_breedPropertyShape>, <Dog_barkPropertyShape>;
14   sh:targetClass <Dog> .
15
16 <Dog> a owl:Class .
```

An inline Schema Object is the most common way to express a Schema Object with polymorphism. However, the response object can be named as well. This is showcased in Listing [20] This has no real impact on the instantiation algorithms:
after replacing all objects referred with $ref the result will be the same with Listing 19

Listing 20. Polymorphed named object.

```
responses: 
  "201": 
    description: variety 
    content: application/json:
      schema:
        oneOf:
          - $ref: "#/components/schemas/TwoPets"
          components:
            schema:
              OneOf:
                - $ref: "#/components/schemas/Cat"
                - $ref: "#/components/schemas/Dog"
```

Polymorphism in property Schemas is also supported. Properties can also take alternative Schemas. Listing 21 shows that property speed can be of any type following keyword anyOf which, in turn, can be defined inline or referred to using keyword $ref. The resulting Shape model is shown in Listing 22 and is composed of one named Shape class (for the property referred to by $ref and two un-named classes for the remaining properties). Keyword oneOf in property Schemas are handled likewise by replacing predicate sh:or in line 4 of Listing 22 with predicate sh:xone.


```
Pet:
  type: object 
  required:
    speed
  properties:
    speed:
      anyOf:
        - type: integer
          format: int64
        - type: string
          $ref: '#/components/schemas/Specified'
  components:
    schemas:
      Specified:
        type: integer
        format: int32
      ... 
```

Listing 22. Polymorphed property Shape model.

```
<Pet_speedPropertyShape> a sh:PropertyShape ; 
  rdfs:label "Pet_speedPropertyShape" ; 
  openapi:name "speed" ; 
  sh:or {<SpecifiedPropertyShape> 
    sh:datatype xsd:string 
  } 
  [ a sh:PropertyShape ; 
    sh:datatype xsd:long 
  ] 
```
Keyword *not* is used in property Schemas to express not acceptable values (i.e., it makes Schemas more specific). For example, property `byType` in Listing 23 can be anything but a string. The SHACL constraint is expressed with predicate `sh:not`. The resulting Shape model in Listing 24 denotes that the corresponding PropertyShape may not take any of the values specified in the RDF list. The restricted property cannot be annotated (e.g., by adding `x-refersTo: https://example.com/ontology/SpecificString` after line 10 in Listing 23). This would allow every other string but the one referred to by the annotation. This negates the universality of the *not* keyword and its original purpose.

Listing 23. Restricted property schema.

```json
Pet:
  type: object
  required:
    - id
  properties:
    id:
      type: integer
      format: int64
    byType:
      type: string
      not:
        type: string
```


```json
<Pet_byTypePropertyShape> a sh:PropertyShape ;
  rdfs:label "Pet_byTypePropertyShape";
  openapi:name "byType";
  sh:not [ a sh:PropertyShape ;
    sh:datatype xsd:string ] ;
  sh:path <Pet_byType> .
<Pet_byType> a rdf:Property .
```

Semantically annotating polymorphed property Schemas is beneficial and not at all pointless. A property Schema with polymorphism basically means that it accepts alternative datatypes. There are two cases. The first is when a semantic value is connected with different Schemas through a property. The second is when a property is connected with semantic values through property attributes (i.e., properties of properties). Both categories are introduced in property Schemas with polymorphism support along with semantic annotations. When a property is annotated with
a semantic value, the Schema property and the semantic value are connected. For example, if the annotation `x-refersTo: https://example.com/ontology/Speed` is inserted between lines 6 and 7 in Listing 21, the property can be of any listed after keyword `anyOf` (or after keyword `oneOf` in a similar example). In fact, the annotation is used to connect the Schema of property `speed` with the Schemas in the RDF list in Listing 22. The second case is when the property attributes are semantically annotated instead of the property itself. For example, if any of the properties following keyword `anyOf` (equivalently `oneOf`) is annotated, property `speed` does not acquire a semantic value but its attributes do. This is showcased in Listing 25. In Listing 26, the Shape model of `speed` is presented along with the semantic value of each of the nodes.

Listing 25. Polymorphism with annotated property Schemas.

```plaintext
Pet:
    type: object
    required:
    - speed
    properties:
    speed:
        anyOf:
        - type: integer
          format: int64
          x-refersTo: https://example.com/ontology/Int64_property
        - type: string
          x-refersTo: https://example.com/ontology/String_property
        ...
components:
    schemas:
    Specified:
        x-refersTo: https://example.com/ontology/Int32_property
        type: integer
        format: int32
```

Listing 26. Polymorphed Shape models with annotated property Shape Schemas.

```plaintext
Pet_speedPropertyShape > a sh: PropertyShape ;
    rdfs: label "Pet_speedPropertyShape";
    openapi: name "speed";
    sh: or ( [ a sh: PropertyShape ;
        sh: datatype xsd: string ;
        sh: path <https://example.com/ontology/String_property>
    ] [ a sh: PropertyShape ;
        sh: datatype xsd: long ;
        sh: path <https://example.com/ontology/Int64_property>
    ] ) .
<SpecifiedPropertyShape> a sh: PropertyShape ;
    sh: datatype xsd: int ;
```

These are the only examples that represent a valid semantically annotated property Schema with polymorphism. There are also cases that are invalid (i.e. may
An invalid case of such property is when all of the components as well as the main property have a semantic value (e.g. when an annotation like \texttt{x-refersTo:https://example.com/ontology/Speed} is added between lines 6 and 7 in Listing 25). The contradiction here is that a semantic value is associated with other semantic values, which can be inconsistent. In this case, the algorithm will exit with the appropriate error message.

5. Ontology Evaluation

Ontology evaluation methods\cite{onteval} investigate the extent to which an ontology conforms to the criteria of accuracy, completeness, conciseness, adaptability, clarity, and consistency. The OpenAPI ontology conforms to the above standards to the extent that OpenAPI specification does. It supports almost the entire range of OpenAPI objects and the properties foreseen in the specification, with the exception of Webhooks, Links, and Callbacks (i.e. left for future work). Evaluating a complex qualitative ontology is not easy. A detailed evaluation of the OpenAPI ontology would require a team of expert users to run an exhaustive set of tests (also left as future work). Task-based ontology evaluation approaches consider also the criterion of computational efficiency: an ontology is evaluated based on its performance in a specific task. In the following, the evaluation of the OpenAPI ontology is mainly focused on the efficiency of ontology translation and query searching.

The ontology translation process has been incorporated into a Web application\cite{ontapp}. The main functionality of the application supports uploading OpenAPI descriptions of REST services (in YAML or JSON format) and their instantiation to the ontology. The application is built using a microservice architecture\cite{microservices} and consists of 3 main services. The front-end service provides a simple user interface for the user to interact with the service. The semantic service acts as the middle layer between the front end and the triple store. It is responsible for the conversion of service descriptions into ontologies and the validation and execution of SPARQL queries. The semantic service provides a REST API which is used exclusively by the front-end service. Finally, the triple store service consists of a Virtuoso Universal Server\cite{virtuoso} acting as a database for persisting and querying the generated ontologies.

The translation of OpenAPI service descriptions to the ontology is not a real-time process. It takes more than 1 second for an OpenAPI service description (i.e. YAML file with more than 200 lines) to translate to ontology and store it in Virtuoso. Table 4 reports the average sizes of the OpenAPI files and of their corresponding triple store files. Notice that, the ontology schema triples are replicated in every ontology. However, this information has no value to the querying process. The reason for including this in the storage is the requirement of Virtuoso which (prior to storage) runs the reasoner to check for inconsistencies and infer new relations from

\begin{itemize}
  \item \cite{ontapp}
  \item \cite{microservices}
  \item \cite{virtuoso}
\end{itemize}
existing ones. An obvious optimization would be to store (for each ontology) only statements that describe the individuals of an OpenAPI document.

<table>
<thead>
<tr>
<th>Table 4. Average size of OpenAPI and of corresponding ontology files.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average size (number of lines)</td>
</tr>
<tr>
<td>OpenAPI file (YAML)</td>
</tr>
<tr>
<td>200</td>
</tr>
<tr>
<td>500</td>
</tr>
<tr>
<td>850</td>
</tr>
<tr>
<td>Ontology Schema (TTL format)</td>
</tr>
<tr>
<td>2,180</td>
</tr>
<tr>
<td>2,170</td>
</tr>
<tr>
<td>2,190</td>
</tr>
<tr>
<td>Ontology Individuals (TTL format)</td>
</tr>
<tr>
<td>360</td>
</tr>
<tr>
<td>580</td>
</tr>
<tr>
<td>830</td>
</tr>
</tbody>
</table>

More than 300 OpenAPI service descriptions have been downloaded from SwaggerHub\(^9\) and translated to ontologies. The generated ontologies are persisted in the triple store of the Web Application. The user may search the triple store for services by submitting SPARQL queries. As a use case scenario, a developer is looking for a Web service by searching for a semantic value that is related to the relevant Web Service description. The SPARQL query of Listing 27 requests all services and operations related to a specific semantic concept schema:Pet. Because the reasoner is invoked prior to storage, the query will also retrieve services related to all its subclass concepts (i.e. schema:Lizard). Therefore, the OpenAPI ontology can be used for improving the performance of a search engine over a collection of OpenAPI descriptions (e.g. SwaggerHub). To improve the efficiency of queries a dedicated query language has been designed\(^20\). The performance results reveal that the performance of queries depends also on query complexity and specificity. Query optimization and indexing on commonly used properties might improve the performance (left as future work).

Listing 27. Example query on hidden relationships.

```sparql
PREFIX openapi: <http://www.intelligence.tuc.gr/ns/open-api>
PREFIX product: <http://www.productontology.org/doc/>

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

Table 5 illustrates the response time of two more indicative SPARQL queries and shows that the response time increases as queries become more complicated.

\(^9\)https://app.swaggerhub.com/
### Table 5. Response time of SPARQL queries on the OpenAPI ontology data set.

<table>
<thead>
<tr>
<th>Simple query</th>
<th>Complex query</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT ?servTitle ?code ?opName WHERE {</td>
<td>SELECT ?servTitle ?code ?opName WHERE {</td>
</tr>
<tr>
<td>?info openapi:serviceTitle ?servTitle .</td>
<td>?info openapi:serviceTitle ?servTitle .</td>
</tr>
<tr>
<td>}</td>
<td>{ ?shape sh:targetClass purl:Pet . }</td>
</tr>
<tr>
<td></td>
<td>UNION { ?shape sh:targetClass purl:Pet . }</td>
</tr>
<tr>
<td></td>
<td>OPTIONAL {</td>
</tr>
<tr>
<td></td>
<td>?tag a openapi:Tag .</td>
</tr>
<tr>
<td></td>
<td>?tag openapi:name ?tgName .</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>OPTIONAL {</td>
</tr>
<tr>
<td></td>
<td>?service openapi:supportedOperation ?operation .</td>
</tr>
<tr>
<td></td>
<td>?server a openapi:Server .</td>
</tr>
<tr>
<td></td>
<td>?server openapi:host ?serverUrl .</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>Get all operations for every service as well as the described status codes</td>
<td>Get all services with operations having responses referring on specific semantic concepts</td>
</tr>
<tr>
<td>Response time: 2,193 ms</td>
<td>Response time: 8,341 ms</td>
</tr>
</tbody>
</table>

### 6. Conclusions and Future Work

Building upon the newest version of the specification, this work analyses the reasons that cause ambiguities in OpenAPI service descriptions. In order to eliminate ambiguities, this work suggests that OpenAPI properties must be semantically annotated and shows how service descriptions can be instantiated to an ontology. Leveraging the latest results for hypermedia-based construction of Web APIs, service descriptions were translated to the OpenAPI ontology. Instantiating OpenAPI descriptions of REST APIs to an ontology is also a problem this work is dealing with. Instantiating Schema objects (i.e. reusable objects which are used by other OpenAPI objects in order to explain their meaning) is a key issue. Taking advantage of composition and polymorphism keywords of OpenAPI, this work introduces the concepts of polymorphism and inheritance to the resulting ontology thus allowing users and applications to take advantage of semantic and machine-readable representations of service descriptions using standard Semantic Web tools such as reasoners and query languages. The instantiation process has been incorporated into an application that is available on the Web for testing.

Future improvements to the ontology and algorithm will incorporate comments received by its users worldwide. Query formulation on the catalog can be facilitated by Graphical User Interfaces allowing users to select service properties from the
pull-down menu. A dedicated query language in the spirit of SOWL-QL \cite{karavisileiou2020ontology} will be designed so that the user need not be familiar with the peculiarities and syntax of the underlying representation. Ongoing work focuses on the Internet of Things domain and shows how OpenAPI can be applied to provide semantically representation for Things (e.g. IoT devices) based on JSON along with a set of RESTful services that enable access of Things on the Web \cite{bouraimis2021instantiating}. Enabling automatic service synthesis and orchestration is a long and more ambitious goal of this approach. The ontology is currently being extended to support the translation of Links and Callbacks OpenAPI properties which are introduced in the latest update of the specification. Callbacks are asynchronous requests that the server service will send to some other service in response to certain events. This feature improves the workflow that the server API offers to its clients. Links, enable the description of how various values returned by one operation can be used as input for other operations. Both these new features will bring more support to HATEOAS in the ontology.

Acknowledgment

We are grateful to Google for the Google Cloud Platform Education Grants program. We also thank Mr. Nikolaos Lagogiannis for preparing the OpenAPI ontologies database and testing the Web application.

References


URL https://www.w3.org/TR/wsd1/


URL https://www.w3.org/TR/sawsdl/


URL https://link.springer.com/chapter/10.1007/978-3-7643-8575-0_3


URL https://www.w3.org/Submission/wadl/

13. J. de Bruijn at al., Web Service Modeling Ontology (WSMO), w3C Member Submission (Jun. 2005).

URL https://www.w3.org/Submission/WSMO/

14. D. M. at al., OWL-S: Semantic Markup for Web Services, w3C Member Submission (Nov. 2004).

URL https://www.w3.org/Submission/OWL-S/


URL https://dl.acm.org/doi/10.5220/0005591001790186


URL https://dias.library.tuc.gr/view/93679


