Technical University of Crete

Diploma Thesis

Description of Asynchronous Service Functionality in the OpenAPI Standard

---

Author:
Emmanouil-Georgios Ieronymakis

Committee:
Euripides G.M. Petrakis
Chrysa Tsinaraki
Georgios Chalkiadakis

A thesis submitted in fulfillment of the requirements for the degree of 5-year Diploma

July 30, 2024

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

Description of Asynchronous Service Functionality in the OpenAPI Standard

by Emmanouil-Georgios Ieronymakis

Publishing service descriptions on the web is critical to their discovery and dissemination in the marketplace. The OpenAPI Specification is a powerful framework for HTTP(S) and RESTful services, endorsed by the Linux Foundation and supported by major software vendors such as Google and Microsoft. OpenAPI comprises a large set of properties for composing service descriptions. The syntactic binding of OpenAPI format to JSON (or YAML) complicates the detection of similarities, inconsistencies, or ambiguities in service descriptions. A previous work introduced, an OpenAPI Ontology for REST Services that emphasized the mapping of Schema properties and the ways they are combined with other properties to form composed or polymorphic expressions in an ontology. This work extends and integrates previous work efforts on mapping OpenAPI descriptions to an ontology. It adds new functionality and implements a more efficient ontology translation mechanism for mapping complex Schema objects and the asynchronous features of the latest OpenAPI version (i.e., Links, Webhooks, and Callbacks). The process has been assessed both qualitatively and quantitatively. The qualitative evaluation guarantees the structural and semantic integrity of the ontology. The qualitative evaluation supports our claim of real-time efficiency for both the ontology mapping and the query search on a triple-store repository using 10,000 OpenAPI descriptions downloaded from Swaggerhub.

Acknowledgements

I would like to express my gratitude to my Supervisor, Professor Euripides G.M. Petrakis for his guidance and support throughout this work. I would also like to thank Nikolaos Mainas and Chrysa Tsinaraki for their key role in shaping the direction and quality of this work.

Last but not least I want to thank my family for supporting me through all these years.

Contents

List of Figures

Chapter 1

Introduction

In today's rapidly evolving digital landscape, application programming interfaces (APIs) of services facilitate data exchange and system integration. API descriptions provide consistent documentation of the role and interaction of services with other services. OpenAPI is the state-of-the-art approach to providing human and machine-readable descriptions in JSON (or YAML) to enable the discovery of services in repositories or on the Web. It also provides a comprehensive set of tools for developers to document or create APIs in a user-friendly way. This standardization enhances user collaboration and improves overall quality and user experience.

OpenAPI presents challenges when it comes to the interpretation of JSON (or YAML) service descriptions by machines. OpenAPI provides a structured way to define API endpoints, datatypes, and other elements, but the semantics of certain operations or schemas may still be complex. Machines may struggle to grasp the intended meaning, especially if there are relationships or conditional logic embedded in the specification. Describing security requirements and mechanisms in OpenAPI can be challenging for machines, especially when dealing with complex security protocols. Furthermore, the quality and consistency of descriptions can vary. Inconsistencies or deviations from the standard may hinder machine interpretation. A way around this is to use rich semantic descriptions of services.

OpenAPI services are published in Web service repositories by their software vendors. The advantages of using OpenAPI are many: it is a complete framework supported by a large set of tools including the 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. An ontology will enable machines to decipher the semantics of OpenAPI, which results in the development of more intelligent and context-aware applications. It is 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.

Instead of developing a new ontology, we choose to map the OpenAPI descriptions to an ontology that is compatible with the original OpenAPI descriptions. OpenAPI and OpenAPI ontology are complementary representations and do not replace one another. Service developers don't change their style of work. They can still use OpenAPI to describe services in JSON (or YAML) in their everyday routine work. The system developed in this work automatically translates OpenAPI descriptions to ontology individuals to assist in developing consistent API descriptions. Neither do they need to work on the peculiarities and syntax of the ontology. In addition, the ontology should not be necessarily visible to the end user or API developer. The ontology settles in the background and its role is to: (a) inform the user of possible ambiguities and inconsistencies in OpenAPI descriptions and (b) work as a service repository and allows to discover similarities between Web APIs, and (c) provide a user interface to allow searching in OpenAPI repositories using Semantic Web Query languages.

1.1 Contributions of the Work

This work summarizes, extends, and integrates previous work efforts of the Intelligent System Laboratory members towards ontology descriptions of OpenAPI. Developing an enhanced OpenAPI Ontology with advanced features and a complete algorithm for instantiating the full set of OpenAPI v3.0 and the most essential features of OpenAPI v3.1 representations is the main focus of this work. More specifically, this work proposed a new enhanced OpenAPI ontology with advanced features that include (a) representation of OpenAPI asynchronous features (i.e., features introduced in OpenAPI v3.0 such as Callabacks, Links, and WebHooks), (b) improvements on annotated Schema object handling, (c) a revamped algorithm designed to seamlessly map OpenAPI descriptions to the newly introduced ontology, and (d) validation of the source JSON (or YAML) file for compatibility with the OpenAPI specification before ontology translation. OpenAPI descriptions are typically created by humans, that are prone to syntax errors and inconsistent or incomplete object and property definitions.

The primary objective of this work is to describe the asynchronous features of OpenAPI (Webhooks, Callbacks) using semantic web technologies and seamlessly integrate these descriptions into the OpenAPI ontology in a coherent manner. Similar to the previous version [14], the new ontology incorporates Hydra [13] concepts for modeling service operations and adds new models for REST asynchronous functionalities such as Callbacks, Links and Webhooks. The revamped ontology instantiation algorithm handles all OpenAPI objects and their properties (e.g. paths, operations, callbacks, service webhooks, links, and re-usable schemas).

This work's efforts concentrate on refining the previous algorithm for mapping and transforming OpenAPI objects and properties into ontology individuals. This revamp aims to enhance data representation and efficiency, with a significant focus on improving the transformation of schema objects and their semantics. The run-time efficiency of the ontology instantiating method has been assessed experimentally on a large set of 10,000 real-world OpenAPI service descriptions downloaded from the Swaggerhub. A significant effort was made to validate the ontology against structural and semantic errors in the OpenAPI sources. Since OpenAPI files are (mainly) composed by humans, their content can be incomplete (e.g., objects or object properties can be missing), inconsistent or ambiguous (i.e., property data types can be incorrect). Ontology mapping is a complex process involving the mapping of several objects. To guarantee ontology integrity, in the presence of errors, besides warning the client, no partial object mappings are accepted (i.e. the whole mapping process is atomic).

This extensive validation experiment also included tests with SQL queries on top of 10,000 OpenAPI (JSON) files stored in a MongoDB using [1] and SPARQL queries on the OpenAPI ontology stored in a GraphDB. Performance comparisons revealed surprisingly good results for complex SPARQL queries (i.e., their performance can be as good as that of SQL queries if the same queries are issued on a MongoDB).

Additionally, some key differences between the new and old version of the ontology are analyzed below. In the old version of the ontology a *Parameter Class* exists, which contains two subclasses *PathParameter Class* and *QueryParameter* Class. In the new version *Parameter Class* has been removed which results in no relationship between the *PathParameter Class* and *QueryParameter Class*. In the License Class the property *spdxIdentifier* has been added in order to describe the license type with this specific format. In *Security Class* two new "security types" have been added: The *MutualTLS Class* in order to describe the new security scheme added in OpenAPI 3.1 and the *NoSecurity Class*, which describes all the API endpoints that have no security. We have also added the *HttpAuthScheme Class* in order to describe all the security schemes that the HttpSecurity Class can support.

Related work on service description languages (with emphasis on REST services) is presented in Chapter 2. The OpenAPI specification and the annotation method used for resolving ambiguities in service descriptions are discussed in Section 3. The OpenAPI ontology and the instantiation of semantic service descriptions compliant to the ontology are discussed in Section 4. Ontology evaluation is described in Section 8, followed by conclusions and future work directions in Section 9.

Chapter 2

Related work

OpenAPI Specification[1], 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 proposed approach and 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 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 [4] for SOAP services, WSDL [4] is an XML-based interface description language for SOAP services [16] (i.e. how and where their functionality can be invoked). The latest version 2.0 introduced changes in the document structure as well as HTPP 1.1 protocol support in order to allow description of REST services. Its previous version is still preferred for the description of SOAP services. SAWSDL [6] defined extension properties in order to add semantics on 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 [9]. WADL [7] 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

[1]https://www.openapis.org

service resources. RAML[2] and API Blueprint [3] 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 [5] defines a conceptual model and WSML language for the semantic description of Web services. OWL-S [2] is an upper ontology for Web services but, similar to all other methods, do not support the dynamic discovery of resources at runtime. SA-REST [21] 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-understadable. Hydra [13] 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 [22] 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.

Musyaffa et al. [18] 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 [19] 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 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 hypermedia-driven Web services like REST. Finally, they do not handle OpenAPI v3.0 descriptions. In a recent contribution, Hamza et al. [8] 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.

Despite its rigorous service language format (JSON or YAML), OpenAPI service descriptions can be vague [14]. There can be properties that share

[2]https://raml.org
[3]https://apiblueprint.org

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. For a machine to act similarly to a human, it is necessary to provide additional information. To eliminate ambiguities, OpenAPI properties must be semantically annotated and associated with entities of a semantic model (e.g. using *Schema.org* vocabulary) using the extension properties.

Besides strong theoretical foundations, semantic approaches for the Web of Things are not well-accepted by the industry and have not been used in real-world or large-scale implementations. 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. 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. Also, information completeness (that relates to ontology size and complexity) and speed of processing, are traded-offs.

More recent approaches attempt to fill the expression gap between syntactic and semantic approaches for the description of REST services. For example, SA-REST [20] offers a set of properties for annotating service descriptions in HTML. Similarly, Musyaffa et al. [18] introduce annotations in *Schema* and *Parameter* objects for OpenAPI v2.0 but, they do not handle all causes of ambiguity. The annotations appear within text properties and cannot be interpreted by a machine without pre-processing.

More recent works attempt to map OpenAPI to ontologies. Schwichtenberg, Gerth, and Engels [19] map OpenAPI v2.0 service descriptions to the OWL-S ontology but, again, they do not deal with any causes of ambiguity, nor do they handle security properties. The mapping is error-prone and must be adjusted manually. In both these works, the choice of OWL-S for representing REST services is controversial: OWL-S is suitable for SOAP services but not for hypermedia-driven Web services like REST.

Muhamad et al. [17] present an ontology for OAS version 3.0. Objects from the OpenAPI document are parsed and saved into appropriate tables of a relational database. When data extraction from the OpenAPI document is complete, database tables are transformed into RDF triples forming an ontology. In contrast to our approach, their work does not incorporate the widely accepted Hydra [13] vocabulary. In addition, the mapping of OpenAPI objects and properties to ontology components is not explained in depth. Based on their article, their implementation lacks support for service and operation security objects, a feature that we have incorporated in our work. Also, the handling and translation of schema objects when they contain keywords (oneOf, anyOf, allOf, not) is not described.

In a recent work [14] we proposed a reference ontology for OpenAPI descriptions based on Hydra [13]. The OpenAPI ontology incorporates the majority of Hydra concepts for modeling service operations and adds new models for REST security, headers and constraints. Classes together with constraints on class properties are described using SHACL [10], allowing service descriptions to be validated against the ontology. The ontology instantiation algorithm [3] handles most OpenAPI objects and their properties (e.g. paths, operations, and re-usable objects such as security and schemas). At the heart of the 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) and their properties can be combined to form complex or polymorphic expressions.

None of the above-mentioned works support the mapping of features introduced in the latest OpenAPI version (i.e. complex Schema objects and asynchronous features such as Linke, Webhooks, and Callbacks), and their performance has not been assessed using a large dataset. Finally, their resulting ontology has not been verified for structural and semantic integrity.

Chapter 3

OpenAPI

OpenAPI service descriptions comprise many objects[1]. Each object has a list of properties that can be objects as well. Objects and properties defined under the *Components* section of an OpenAPI document can be re-used by other objects or they can be linked to each other (e.g. using the *$ref* keyword). However, these links are not always explicitly expressed (i.e. there are properties with the same name with no reference to one another or 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 describes where the API servers are located (i.e. multiple servers can be defined). The *Security* object describes the security schemes that the service uses for authentication (i.e. API keys, OAuth2.0 common flows, and OpenID Connect). The *Paths* object 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 information for expressing HTTP requests to the service and its responses which are typically defined under the *Components* section of reusable objects. These objects can be responses, parameters, schemas, request bodies, and more. 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 *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 *Schema* object specifies the data type that describes the request and response messages. It can be a primitive (i.e. string, integer), an array, or a model. For the definition of *Schema* objects, the specification resorts to JSON Schema[2]. A *Schema* object definition can be enhanced with XML data types. New data types can be defined as a combination of existing ones (i.e. using the *allOf*, or the *oneOf* property).

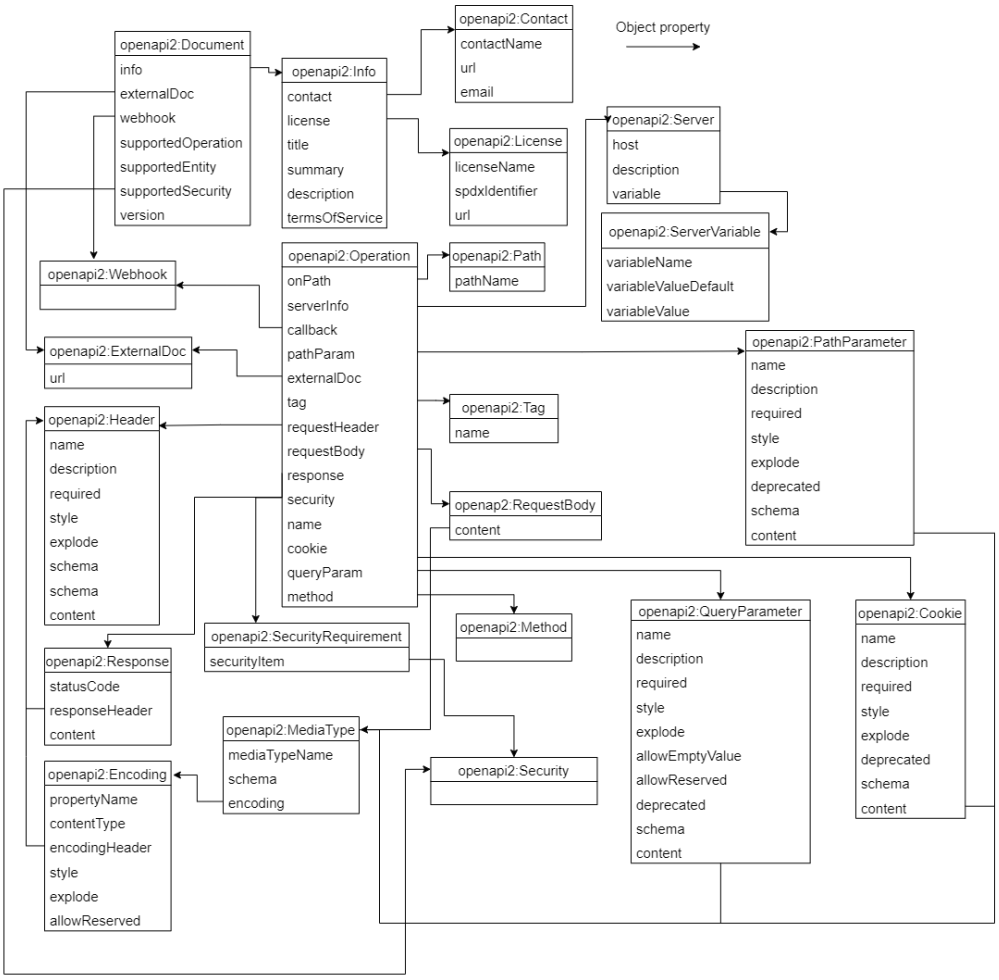
[1] https://blog.readme.io/an-example-filled-guide-to-swagger-3-2/
[2] http://json-schema.org

Chapter 4

OpenAPI Ontology

The following is an overview of the OpenAPI ontology. Building upon the Hydra core vocabulary [13], the ontology covers almost every aspect of the whole OpenAPI specification version 3.1. This includes a comprehensive mapping of its most recent elements such as Webhooks, Callbacks, and Links, but it also includes translations of the previous versions. Optimizing the mapping of Schema Objects and their respective properties required significant effort. OpenAPI v3.1 format claimed full compatibility with JSON Schema. However, the translation of the full set of JSON Schema objects is not possible (i.e. interpretation and validation of every intended use of any JSON schema is open-ended). Apart from this, the OpenAPI ontology is very robust. Figure 4.1 is an overview of the OpenAPI ontology. Each object specifies a list of properties that can be objects as well.

FIGURE 4.1: OpenAPI ontology diagram (simplified)



The **Document class** represents documentation about the service. The *Info* class provides contact, license, and terms of service information. The **Path class** represents (relative) service paths (i.e. the *pathName* property). The **Operation Class** is used to describe the service-supported REST operations. The **Webhook class** describes asynchronous requests (to another service) initiated from the service that does not rely on an API call. The Webhook class is also used to describe **Callbacks** which are also asynchronous requests that are initiated when some type of event occurs. Request bodies are represented by the **RequestBody class**. Responses are declared in the **Response class** that specifies the status code and the data returned. The **MediaType class** describes the format of request or response body data (the most common being JSON and XML).

Parameters are represented by a separate class for each 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 **PathParameter** and **QueryParameter classes** refer to the corresponding path and query parameters of the specification. The **Encoding class**

defines keywords denoting serialization rules for media types with primitive properties (e.g. *contentType* for nested arrays or JSON).

Figure 4.2 illustrates classes representing the OpenAPI security schemes. The Operation class refers to a security scheme in the SecurityRequirement class. The **Security class** is a generic class, which describes the security schemes that can be supported by an API and it is further detailed in the following security scheme subclasses:

- **ApiKeySecurity class**, which describes the API key security scheme
- **HttpSecurity class**, which describes the HTTP authentication scheme
- **OAuth2 class**, for the OAuth2 common flows
- **OpenIdConnect class**, which describes the OpenID Connect discovery
- **MutualTLS class**, describing mutual TLS security
- **NoSecurity class**, describing the absence of security

4.1 *Transforming OpenAPI service descriptions into OpenAPI ontology individuals*

Algorithm 1 summarizes the ontology instantiation process. Initially, a *Document class* individual is created representing the service. It also acts as a 'root' individual for the resulting service description. The algorithm proceeds with the translation of reusable objects in the components section (lines 4-13). Reusable Schema objects are converted to individuals of *SHACL Shape classes* [10]. Parameter, and RequestBody objects are converted into *PathParameter/QueryParameter/Cookie/Header Class* and *RequestBody class* individuals. These individuals are cached for future reference.

The Server objects (defined inside the OpenAPI object) are converted to *Server class* individuals (lines 14-16). Strategically stored in the cache, these instances will be ready for use when no servers are explicitly declared at the Operation level. Subsequently, the OpenAPI object's Info and ExternalDocument objects are converted into instances of the *Info Class* and *ExternalDoc Class*, respectively (lines 17-22). Next (lines 23-25) is the translation of reusable SecurityScheme objects defined in the components section into *ApiKeySecurity/HttpSecurity/OAuth2/OpenIdConnect/MutualTLS Class* individuals. OpenAPI's SecurityRequirement objects are converted into *SecurityRequirement Class* individuals (lines 26-28). OpenAPI PathItem object translation takes place in lines 29-30. For each PathItem object, a *Path Class* individual is created. OpenAPI's Tag objects are converted

into *Tag Class* individuals (lines 31-32). Service Webhooks (only in version 3.1 of OpenAPI) defined in the OpenAPI object are converted into *Webhook Class* individuals in lines 33-36. Lastly, Operation objects are converted to Operation Class individuals, by looping through PathItem objects and their supported methods (lines 37-40). More detail on the translation of Webhooks will be given in section 5.

Symbol	Meaning
Object.function()	`A function call`
Object $\rightarrow$ **Property**	`An OpenAPI's object property`
Subject $\xrightarrow{Predicate}$ **Object**	`An RDF Triple`

Table 4.1: Symbol meaning

Algorithm 1 Converting OpenAPI object to ontology

1: **procedure** parseDocument(document)
2: Initialize Ontology Model and Variables
3: serviceIndividual $\xrightarrow{rdf:type}$ openapi2:Document ▷ Initialize Service Individual

4: Fetch Components
5: **for** Schema Object in (Components $\rightarrow$ schemas) **do**
6: schemaIndividual = SchemaTranslator.translate(schema, schemaName)
7: Cache.addSchemaIndividual(schemaName, schemaIndividual)
8: **for** Parameter Object in (Components $\rightarrow$ parameters) **do**
9: parameterIndividual = ParameterTranslator.translate(parameter)
10: Cache.addParameterIndividual(parameterName, parameterIndividual)
11: **for** RequestBody Object in (Components $\rightarrow$ requestBodies) **do**
12: requestBodyIndividual = RequestBodyTranslator.translate(requestBody)
13: Cache.addRequestBodyIndividual(requestBodyName, requestBodyIndividual)

14: **for** Server Object in (OpenAPI Object $\rightarrow$ servers) **do**
15: serverIndividual = ServerTranslator.translate(server)
16: Cache.addRootServer(serverIndividual)

17: Fetch Info Object from OpenAPI Object
18: infoIndividual = InfoTranslator.translate(info)
19: serviceIndividual $\xrightarrow{openapi2:info}$ infoIndividual

20: Fetch ExternalDoc Object from OpenAPI Object
21: externalDocIndividual = ExternalDocTranslator.translate(externalDoc)
22: serviceIndividual $\xrightarrow{openapi2:externalDoc}$ externalDocIndividual

Algorithm 1 Converting OpenAPI object to ontology - Part 2

23: **for** SecurityScheme Object in (Components → securitySchemes) **do**
24: securitySchemeIndividual = SecuritySchemeTranslator.translate(securityScheme)
25: serviceIndividual $\xrightarrow{openapi2:supportedSecurity}$ securitySchemeIndividual

26: **for** SecurityRequirement Object in (OpenAPI Object → security) **do**
27: secReqIndividual = SecurityRequirementTranslator.tranlsate(securityRequirement)
28: Cache.addRootSecurityRequirement(secReqIndividual)

29: **for** PathItem Object in (OpenAPI Object → paths) **do**
30: pathIndividual = PathTranslator.translate(pathItem)

31: **for** Tag Object in (OpenAPI Object → tags) **do**
32: Create Tag individual

33: **for** Webhook PathItem Object in (OpenAPI Object → webhooks) **do**
34: **for** Operation Object in Webhook PathItem Object **do**
35: webhookIndividual = WebhookTranslator.translate(operation, operationMethod, eventName)
36: serviceIndividual $\xrightarrow{openapi2:webhook}$ webhookIndividual

37: **for** PathItem Object in (OpenAPI Object → paths) **do**
38: **for** Operation Object in PathItem Object **do**
39: operationIndividual = OperationTranslator.translate(operation, operationMethod, pathIndividual)
40: serviceIndividual $\xrightarrow{openapi2:supportedOperation}$ operationIndividual

4.2 *Transforming OpenAPI Info Objects into Info Class individuals*

Algorithm 2 outlines the steps for converting an OpenAPI Info object into an Info class individual. The Info object provides metadata about the API which can be used by the clients. The algorithm begins by creating an Info class individual. Moving on, it checks for the basic properties, *title*, *summary*, *description* and *termsOfService*, assigning these properties to the individual (lines 2-10). Subsequently, the algorithm checks for a *Contact Object* declared within the Info Object. If present, a *Contact Class* individual is created, which is then populated with properties obtained from the contact object (lines 11-21). The object of the newly created contact class is linked to the object of the information class via the *openapi2:contact* property (lines 11-21).

Translating a license object into the information object follows the same steps, creating a *License Class* individual which is then associated with the information class individual via the *openapi2:license* property (lines 22-30). Listing 4.1 showcases a simple OpenAPI Info Object, and Listing 4.2 showcases the generated ontology individuals.

Algorithm 2 Info Object Converter

1: **procedure** TRANSLATE(Info info)
2: infoIndividual $\xrightarrow{rdf:type}$ openapi2:Info ▷ Initialize Info Individual

3: **if** info → title not null **then**
4: infoIndividual $\xrightarrow{openapi2:title}$ title
5: **if** info → summary not null **then**
6: infoIndividual $\xrightarrow{openapi2:summary}$ summary
7: **if** info → description not null **then**
8: infoIndividual $\xrightarrow{openapi2:description}$ description
9: **if** info → termsOfService not null **then**
10: infoIndividual $\xrightarrow{openapi2:termsOfService}$ IRI(termsOfService)

11: **if** info → contact not null **then**
12: contactIndividual $\xrightarrow{rdf:type}$ openapi2:Contact ▷ Initialize Contact Individual

13: **if** contact → name not null **then**
14: contactIndividual $\xrightarrow{openapi2:name}$ name
15: **if** contact → url not null **then**
16: contactIndividual $\xrightarrow{openapi2:url}$ IRI(url)
17: **if** contact → email not null **then**
18: contactIndividual $\xrightarrow{openapi2:email}$ IRI(email)
19: **if** contact → name not null **then**
20: contactIndividual $\xrightarrow{openapi2:name}$ name

21: infoIndividual $\xrightarrow{openapi2:contact}$ contactIndividual ▷ Connect Info, Contact individuals

22: **if** info → license not null **then**
23: licenseIndividual $\xrightarrow{rdf:type}$ openapi2:License ▷ Initialize License Individual

24: **if** license → name not null **then**
25: licenseIndividual $\xrightarrow{openapi2:name}$ name
26: **if** license → spdxIdentifier not null **then**
27: licenseIndividual $\xrightarrow{openapi2:spdxIdentifier}$ spdxIdentifier
28: **else if** license → url not null **then**
29: licenseIndividual $\xrightarrow{openapi2:url}$ IRI(url)

30: infoIndividual $\xrightarrow{openapi2:license}$ licenseIndividual ▷ Connect Info, License individuals

LISTING 4.1: OpenAPI Info object example.

```yaml
info:
  title: Swagger Petstore - OpenAPI 3.1
  description: This is a sample Pet Store Server based on
      the OpenAPI 3.1 specification.
  termsOfService: http://swagger.io/terms/
  contact:
    email: apiteam@swagger.io
  license:
    name: Apache 2.0
    url: http://www.apache.org/licenses/LICENSE-2.0.html
  version: 1.0.11
```

LISTING 4.2: Info Class individual resulting from the translation of the OpenAPI Info object of Listing 4.1

```
% Info Class individual

ex:info a openapi2:Info;
    openapi2:title "Swagger Petstore - OpenAPI 3.1";
    openapi2:description "This is a sample Pet Store
       Server based on the OpenAPI 3.1 specification.";
    openapi2:termsOfService <http://swagger.io/terms/>;
    openapi2:contact ex:infoContact;
    openapi2:license ex:infoLicense .

% Contact Class individual

ex:infoContact a openapi2:Contact;
    openapi2:email <mailto:apiteam@swagger.io> .

% License Class individual

ex:infoLicense a openapi2:License;
    openapi2:licenseName "Apache 2.0";
    openapi2:url <http://www.apache.org/licenses/LICENSE
       -2.0.html> .
```

4.3 *Transforming OpenAPI Server Objects into Server Class individuals*

Algorithm 3 describes the mapping of an OpenAPI Server object to a *Server Class* individual. An API can declare one or multiple servers. Each *Server object* represents one of the API servers. Initially, (lines 1-6), the algorithm scans the Server Object for a *url* property and checks for an optional *description* property, assigning the *openapi2:host* and *openapi2:description properties* to the newly created *Server Class* individual. If the server *url* includes variables, a separate *ServerVariable Class* class is created for each one. The *enum*, *default*, *description* (optional) properties retrieved from the ServerVariable object are then assigned to the ServerVariable class object accordingly (lines 7-16). Each individual of the ServerVariable class is ultimately bound to the individual of the Server class using the *openapi2:variable* property. Listing 4.3 is an example OpenAPI Server Object with *url* variables. Listing 4.4 shows the generated individuals.

Algorithm 3 Server Object Converter

procedure TRANSLATE(Server server)
    serverIndividual $\xrightarrow{rdf:type}$ openapi2:Server ▷ Initialize Server Individual

    if server → url not null **then**
        serverIndividual $\xrightarrow{openapi2:host}$ url
    if server → description not null **then**
        serverIndividual $\xrightarrow{openapi2:description}$ description

    for ServerVariable Object in server → variables **do**
        serverVarIndividual $\xrightarrow{rdf:type}$ openapi2:ServerVariable ▷ Initialize ServerVariable Individual

        serverVarIndividual $\xrightarrow{openapi2:variableName}$ name
        if ServerVariable → default not null **then**
            serverVarIndividual $\xrightarrow{openapi2:variableValueDefault}$ default
        for Enum String in ServerVariable → enum **do**
            serverVarIndividual $\xrightarrow{openapi2:variableValue}$ enum
        if ServerVariable → description not null **then**
            serverVarIndividual $\xrightarrow{openapi2:description}$ description

        serverIndividual $\xrightarrow{openapi2:variable}$ serverVarIndividual

Listing 4.3: OpenAPI Server object example.

```
servers:
    url: https://{customerId}.saas-app.com:{port}/v2
    variables:
      customerId:
        default: demo
        description: Customer ID assigned by the service
           provider
      port:
        enum:
          - '443'
          - '8443'
        default: '443'
```

LISTING 4.4: Server Class individual resulting from the translation of the OpenAPI Server object of Listing 4.3

```
% Server Class Individual

ex:server1 a openapi2:Server;
    openapi2:host "https://{customerId}.saas-app.com:{
        port}/v2";
    openapi2:variable ex:serverVariable1, ex:serverVariable2 .

% customerId ServerVariable Class individual

ex:serverVariable1 a openapi2:ServerVariable;
    openapi2:variableName "customerId";
    openapi2:variableValueDefault "demo";
    openapi2:description "Customer ID assigned by the
        service provider" .

% port ServerVariable Class individual

ex:serverVariable2 a openapi2:ServerVariable;
    openapi2:variableName "port";
    openapi2:variableValueDefault "443";
    openapi2:variableValue "443", "8443" .
```

4.4 *Transforming OpenAPI Parameter Objects into OpenAPI ontology individuals*

Algorithm 4 outlines the procedure for transforming OpenAPI Parameter Objects into OpenAPI ontology individuals. A Parameter object describes a single operation parameter. The algorithm handles all four parameter types, namely Cookie, Header, Path, and Query parameters. Accordingly, these are mapped to four distinct classes (i.e. PathParameter, QueryParameter, Cookie, and Header class).

The algorithm begins by examining the *in* property of the parameter object (lines 2-9). Based on its type (i.e. PathParameter, QueryParameter, Cookie, Header), the algorithm instantiates an individual of PathParameter, QueryParameter, Cookie, or Header class. Following this, the algorithm processes the properties *name*, *description*, *required* of the parameter object (lines 10-17) and also assigns these properties to the individual.

For the *style* property of a parameter object, the ontology introduces seven classes representing each style (i.e., *matrix*, *label*, *form*, *simple*, *spaceDelimited*, *pipeDelimited* and *deepObject* class). The algorithm checks the value of the style property and assigns the *openapi2:style* property to the individual (lines 18-37), using the appropriate style class. For instances with no value for the style property, default values are employed, adhering to the OpenAPI specification. Following this, the algorithm addresses the *explode* property of the parameter object (lines 38-47), assigning the corresponding *openapi2:explode* property to the individual. If no value is provided, default value is used.

Moving to the parameter object's *schema* property, the algorithm checks for a reference to a reusable schema within the components section. If there is a reference, the algorithm retrieves the already transformed individual from the cache. If there is no reference and the schema object is defined inline, the algorithm will convert the Schema Object to a NodeShape or PropertyShape and assign it to the Parameter individual using the *openapi2:schema* property. For parameters of type query, the algorithm additionally handles the properties *allowEmptyValue* and *allowReserved* by assigning the *openapi2:allowEmptyValue* and *openapi2:allowReserved* properties to the parameter individual. Lastly, the *openapi2:deprecated* property is assigned based on the deprecated property value of the parameter object. An example of a Parameter Object is shown in Listing 4.5. The ontology individual generated is shown in 4.6.

Algorithm 4 Parameter Object Converter

1: **procedure** TRANSLATE(Parameter parameter)
2: **if** parameter → in = "path" **then** ▷ Initialize Parameter Individual
3: parameterIndividual $\xrightarrow{rdf:type}$ openapi2:PathParameter
4: **else if** parameter → in = "query" **then**
5: parameterIndividual $\xrightarrow{rdf:type}$ openapi2:QueryParameter
6: **else if** parameter → in = "cookie" **then**
7: parameterIndividual $\xrightarrow{rdf:type}$ openapi2:Cookie
8: **else if** parameter → in = "header" **then**
9: parameterIndividual $\xrightarrow{rdf:type}$ openapi2:Header

10: **if** parameter → name not null **then**
11: parameterIndividual $\xrightarrow{openapi2:name}$ name
12: **if** parameter → description not null **then**
13: parameterIndividual $\xrightarrow{openapi2:description}$ description
14: **if** parameter → in = "path" or parameter → required = true **then**
15: parameterIndividual $\xrightarrow{openapi2:required}$ xsd:true
16: **else**
17: parameterIndividual $\xrightarrow{openapi2:required}$ xsd:false

18: **if** parameter → style not null **then**
19: **if** style = "MATRIX" **then**
20: parameterIndividual $\xrightarrow{openapi2:style}$ openapi2:matrix
21: **else if** style = "LABEL" **then**
22: parameterIndividual $\xrightarrow{openapi2:style}$ openapi2:label
23: **else if** style = "FORM" **then**
24: parameterIndividual $\xrightarrow{openapi2:style}$ openapi2:form
25: **else if** style = "SIMPLE" **then**
26: parameterIndividual $\xrightarrow{openapi2:style}$ openapi2:simple
27: **else if** style = "SPACEDELIMITED" **then**
28: parameterIndividual $\xrightarrow{openapi2:style}$ openapi2:spaceDelimited
29: **else if** style = "PIPEDELIMITED" **then**
30: parameterIndividual $\xrightarrow{openapi2:style}$ openapi2:pipeDelimited
31: **else if** style = "DEEPOBJECT" **then**
32: parameterIndividual $\xrightarrow{openapi2:style}$ openapi2:deepObject
33: **else**
34: **if** parameter → in = "path" or parameter → in = "header" **then**
35: parameterIndividual $\xrightarrow{openapi2:style}$ openapi2:simple
36: **else if** parameter → in = "query" or parameter → in = "cookie" **then**
37: parameterIndividual $\xrightarrow{openapi2:style}$ openapi2:form

Algorithm 4 Parameter Object Converter - Part 2

38: **if** parameter → explode not null **then**
39: **if** parameter → explode = true **then**
40: parameterIndividual $\xrightarrow{openapi2:explode}$ xsd:true
41: **else**
42: parameterIndividual $\xrightarrow{openapi2:explode}$ xsd:false
43: **else**
44: **if** parameter → in = "path" or parameter → in = "header" **then**
45: parameterIndividual $\xrightarrow{openapi2:explode}$ xsd:false
46: **else if** parameter → in = "query" or parameter → in = "cookie" **then**
47: parameterIndividual $\xrightarrow{openapi2:explode}$ xsd:true

48: **if** parameter → schema not null **then**
49: **if** parameter → $ref not null **then**
50: refShapeIndividual = cache.getShapeIndividual($ref)
51: parameterIndividual $\xrightarrow{openapi2:schema}$ refShapeIndividual
52: **else**
53: shapeIndividual = SchemaTranslator.translate(schema)
54: parameterIndividual $\xrightarrow{openapi2:schema}$ shapeIndividual

55: **if** parameter → in = "query" **then**
56: **if** parameter → allowEmptyValue not null and parameter → allowEmptyValue = true **then**
57: parameterIndividual $\xrightarrow{openapi2:allowEmptyValue}$ xsd:true
58: **else**
59: parameterIndividual $\xrightarrow{openapi2:allowEmptyValue}$ xsd:false
60: **if** parameter → allowReserved not null and parameter → allowReserved = true **then**
61: parameterIndividual $\xrightarrow{openapi2:allowReserved}$ xsd:true
62: **else**
63: parameterIndividual $\xrightarrow{openapi2:allowReserved}$ xsd:false

64: **if** parameter → deprecated not null and parameter → deprecated = true **then**
65: parameterIndividual $\xrightarrow{openapi2:deprecated}$ xsd:true
66: **else**
67: parameterIndividual $\xrightarrow{openapi2:deprecated}$ xsd:false

Listing 4.5: OpenAPI Parameter object example.

```
parameters:
    name: token
    in: header
    description: token to be passed as a header
    required: true
    style: simple
```

Listing 4.6: Header Class individual resulting from the translation of the OpenAPI Parameter object of Listing 4.5

```
% token Header Class individual

ex:param1 a openapi2:Header;
    openapi2:name "token";
    openapi2:descriptjon "token to be passed as a header
        ";
    openapi2:required xsd:true;
    openapi2:style openapi2:simple;
    openapi2:explode xsd:false;
    openapi2:deprecated xsd:false .
```

4.5 *Transforming OpenAPI ApiResponse Objects into Response Class individuals*

Algorithm 5 describes the transformation of OpenAPI ApiResponse objects into Response Class individuals in the OpenAPI ontology. An ApiResponse object describes a single response of an API Operation. To explicitly represent the different response types in the ontology we have introduced six distinct classes for responses namely, *DefaultResponse*, *1xxResponse*, *2xxResponse*, *3xxResponse*, *4xxResponse* and *5xxResponse*.

The algorithm begins by examining the *statusCode* value of the ApiResponse object, initiating the creation of a response individual of the appropriate class (lines 2-13). The status code can be provided in the form of "1XX" or as a specific value (e.g. "404"). If the status code has a specific value, an *openapi2:statusCode* property is assigned to the individual with that value. The algorithm also checks for an optional *description* property and assigns the *openapi2:description* property to the individual accordingly.

In addition to the basic response details, an ApiResponse may include headers. The algorithm examines each *Header object* (lines 18-25) declared within the ApiResponse object. A header can be defined either inline or by referencing a reusable one in the components section. If there is a reference, the algorithm retrieves the previously translated header individual from the cache, otherwise it converts the inline Header object. Each header individual is then linked to the response individual using

the *openapi2:responseHeader* property. A MediaType object can be defined within the response object. The algorithm adeptly converts this object and establishes a connection with the response object using the *openapi2:content* property (lines 26-29). An example of an ApiResponse Object is shown in Listing 4.7. The corresponding ontology individuals are shown in 4.8.

Algorithm 5 Response Object Converter

```
1: procedure TRANSLATE(ApiResponse response, String statusCode)
2:     if statusCode = "default" then      ▷ Initialize responseIndividual
3:         responseIndividual --rdf:type--> openapi2:DefaultResponse
4:     else if statusCode in range [100, 199) then
5:         responseIndividual --rdf:type--> openapi2:1xxResponse
6:     else if statusCode in range [200, 299) then
7:         responseIndividual --rdf:type--> openapi2:2xxResponse
8:     else if statusCode in range [300, 399) then
9:         responseIndividual --rdf:type--> openapi2:3xxResponse
10:    else if statusCode in range [400, 499) then
11:        responseIndividual --rdf:type--> openapi2:4xxResponse
12:    else if statusCode in range [500, 599) then
13:        responseIndividual --rdf:type--> openapi2:5xxResponse

14:    if intValue(statusCode) not null then
15:        responseIndividual --openapi2:statusCode--> statusCode
16:    if response → description not null then
17:        responseIndividual --openapi2:description--> description
```

Algorithm 5 Response Object Converter - Part 2

18: **if** response → headers not null **then**
19: **for** Header Object in (response → headers) **do**
20: **if** header → $ref not null **then**
21: refHeaderIndividual = Cache.getHeaderIndividual($ref)
22: responseIndividual $\xrightarrow{openapi2:responseHeader}$ refHeaderIndividual
23: **else**
24: headerIndividual = HeaderTranslator.translate(header, name)
25: responseIndividual $\xrightarrow{openapi2:responseHeader}$ headerIndividual

26: **if** response → content not null **then**
27: **for** MediaType Object in content **do**
28: mediaTypeIndividual = MediaTypeTranslator.translate(mediaType, name)
29: responseIndividual $\xrightarrow{openapi2:content}$ mediaTypeIndividual

Listing 4.7: OpenAPI ApiResponse objects example.

```
responses:
    '200':
      description: successful operation
      headers:
        X-Rate-Limit:
          description: calls per hour allowed by the user
        X-Expires-After:
          description: date in UTC when token expires
    '400':
      description: Invalid username/password supplied
```

Listing 4.8: Response Class Individuals resulting from the translation of the OpenAPI ApiResponse objects of Listing 4.7

```
% '200' response 2xxResponse Class individual

ex:response1 a openapi2:2xxResponse;
    openapi2:statusCode "200"^^xsd:int;
    openapi2:description "successful operation";
    openapi2:responseHeader ex:header1, ex:header2 .

% X-Rate-Limit Header Individual

ex:header1 a openapi2:Header;
    openapi2:name "X-Rate-Limit";
    openapi2:description "calls per hour allowed by the
       user";
    openapi2:required xsd:false;
    openapi2:explode xsd:false;
    openapi2:deprecated xsd:false;
    openapi2:style openapi2:simple .

% X-Expires-After Header Individual

ex:header2 a openapi2:Header;
    openapi2:name "X-Expires-After";
    openapi2:description "date in UTC when token expires
       ";
    openapi2:required xsd:false;
    openapi2:explode xsd:false;
    openapi2:deprecated xsd:false;
    openapi2:style openapi2:simple .

% '400' response 4xxResponse Class individual

ex:response2 a openapi2:4xxResponse;
    openapi2:statusCode "400"^^xsd:int;
    openapi2:description "Invalid username/password
       supplied" .
```

4.6 *Transforming OpenAPI Media Type Objects into MediaType Class individuals*

A media type object specifies the media type of the payload in an HTTP request or response, offering details about the formatting and interpretation of the data. Algorithm 6 describes the translation of a Media Type Object to an individual of the MediaType Class. The algorithm begins by initializing the MediaType class individual. Each Media Type Object may contain an optional Schema Object which defines the content of the request or response. If a Schema Object exists, then the algorithm will check for a reference to a reusable Schema Object. If there is a reference to a reusable schema then the algorithm will retrieve the previously generated individual for that schema and will assign it to the MediaType class individual using the *openapi2:schema* property. In case the Schema Object is defined inline then the algorithm will translate the Schema Object before assigning it to the MediaType class individual.

A MediaType Object may contain an Encoding Object. The Encoding Object is used to specify how the data should be serialized or encoded,

including details about the content type, style, and explode behavior. According to the OpenAPI Specification, encoding is applied only when a Request Body Object is of *multipart* or *application/x-www-form-urlencoded* media type (lines 11-15). If the encoding object exists, the algorithm will translate it to match the encoding class object created in the MediaType class object using the *openapi v2:encoding* property. Listing 4.9 showcases a MediaType Object encoding definition. Listing 4.10 showcases the resulting individuals.

Algorithm 6 MediaType Object Converter

1: **procedure** TRANSLATE(MediaType mediaType, String name)
2: mediaTypeIndividual $\xrightarrow{rdf:type}$ openapi2:MediaType ▷ Initialize MediaType individual
3: mediaTypeIndividual $\xrightarrow{openapi2:mediaTypeName}$ name

4: **if** mediaType → schema not null **then**
5: **if** schema → $ref not null **then**
6: refShapeIndividual = cache.getShapeIndividual($ref)
7: mediaTypeIndividual $\xrightarrow{openapi2:schema}$ refShapeIndividual
8: **else**
9: shapeIndividual = SchemaTranslator.translate(schema)
10: mediaTypeIndividual $\xrightarrow{openapi2:schema}$ shapeIndividual

11: **if** name = "multipart/form-data" or name = "application/x-www-form-urlencoded" **then**
12: **if** mediaType → encoding not null **then**
13: **for** Encoding Object in (mediaType → encoding) **do**
14: encodingIndividual = EncodingTranslator.translate(encoding)
15: encodingIndividual $\xrightarrow{openapi2:encoding}$ encodingIndividual

For the Encoding Object, the algorithm begins by initializing the Encoding Class individual. After initialization, the algorithm will check the contentType property of the Encoding Object and assign it to the individual using the *openapi2:contentType* property. In lines 6-16 the algorithm will select the appropriate Style Class individual based on the value of the style property of the Encoding Object, and it will assign it to the Encoding individual using the *openapi2:style* property. Finally, the algorithm will check the values of the explode and allowReserved properties of the Encoding Object and it will assign the *openapi2:explode* and *openapi2:allowReserved* properties to the individual based on those values. If one or both of these property values are not explicitly defined, the default values will be used defined by the OpenAPI Specification. Algorithm 7 details the transformation of an Encoding Object illustrating how an Encoding Object is mapped into an individual of the Encoding Class.

Algorithm 7 Encoding Object Converter

1: **procedure** TRANSLATE(Encoding encoding, String propertyName)
2: encodingIndividual $\xrightarrow{rdf:type}$ openapi2:Encoding ▷ Initialize Encoding Individual
3: encodingIndividual $\xrightarrow{openapi2:propertyName}$ propertyName

4: **if** encoding → contentType not null **then**
5: encodingIndividual $\xrightarrow{openapi2:contentType}$ contentType

6: **if** encoding → style not null **then**
7: **if** style = "FORM" **then**
8: encodingIndividual $\xrightarrow{openapi2:style}$ openapi2:form
9: **else if** style = "SPACEDELIMITED" **then**
10: encodingIndividual $\xrightarrow{openapi2:style}$ openapi2:spaceDelimited
11: **else if** style = "PIPEDELIMITED" **then**
12: encodingIndividual $\xrightarrow{openapi2:style}$ openapi2:pipeDelimited
13: **else if** style = "DEEPOBJECT" **then**
14: encodingIndividual $\xrightarrow{openapi2:style}$ openapi2:deepObject
15: **else**
16: encodingIndividual $\xrightarrow{openapi2:style}$ openapi2:form

17: **if** encoding → explode not null **then**
18: **if** explode = true **then**
19: encodingIndividual $\xrightarrow{openapi2:explode}$ xsd:true
20: **else**
21: encodingIndividual $\xrightarrow{openapi2:explode}$ xsd:false
22: **else**
23: **if** encoding → style = "FORM" **then**
24: encodingIndividual $\xrightarrow{openapi2:explode}$ xsd:true
25: **else**
26: encodingIndividual $\xrightarrow{openapi2:explode}$ xsd:false

27: **if** encoding → allowReserved not null **then**
28: **if** allowReserved = true **then**
29: encodingIndividual $\xrightarrow{openapi2:allowReserved}$ xsd:true
30: **else**
31: encodingIndividual $\xrightarrow{openapi2:allowReserved}$ xsd:false

Listing 4.9: OpenAPI MediaType object example.

```
content:
  multipart/form-data:
    encoding:
      historyMetadata:
        contentType: application/xml; charset=utf-8
      profileImage:
        contentType: image/png, image/jpeg
```

Listing 4.10: MediaType & Encoding Class individuals resulting from the translation of the OpenAPI MediaType object of Listing 4.9

```
% multipart/form-data MediaType Class individual

ex:mediaType1 a openapi2:MediaType;
    openapi2:mediaTypeName "multipart/form-data";
    openapi2:encoding ex:encoding1, ex:encoding2 .

% historyMetadata Encoding Class individual

ex:encoding1 a openapi2:Encoding;
    openapi2:propertyName "historyMetadata";
    openapi2:contentType "application/xml; charset=utf
        -8";
    openapi2:style openapi2:form;
    openapi2:explode xsd:true;
    openapi2:allowReserved xsd:false .

% profileImage Encoding Class individual

ex:encoding2 a openapi2:Encoding;
    openapi2:propertyName "profileImage";
    openapi2:contentType "image/png, image/jpeg";
    openapi2:encodingHeader ex:header1;
    openapi2:style openapi2:form;
    openapi2:explode xsd:true;
    openapi2:allowReserved xsd:false .
```

4.7 *Transforming OpenAPI RequestBody Objects into RequestBody Class individuals*

In OpenAPI, a Request Body Object is used to describe the request payload that can be sent to an API endpoint. The object consists of three properties, namely description, required, and content. The algorithm begins by initializing a RequestBody class individual. It checks if a description exists and assigns its value to the individual using the *openapi2:description*. Then it scans the value of the required property and assigns it to the individual using the *openapi2:required* property. Finally, the algorithm loops through the MediaType Objects that may be defined in the *content* property. The algorithm converts each one to a MediaType class individual, and proceeds to associate them with the RequestBody Class individual using the *openapi2:content* property. An example of a Request Body Object is shown in Listing 4.11. The ontology individuals

generated are shown in Listing 4.12.

Algorithm 8 RequestBody Converter

procedure TRANSLATE(RequestBody requestBody)
    requestBodyIndividual $\xrightarrow{rdf:type}$ openapi2:requestBody ▷ Initialize RequestBody Individual
    if RequestBody → description not null **then**
        requestBodyIndividual $\xrightarrow{openapi2:description}$ description
    if RequestBody → required not null AND RequestBody → required = true **then**
        requestBodyIndividual $\xrightarrow{openapi2:required}$ xsd:true
    else
        requestBodyIndividual $\xrightarrow{openapi2:required}$ xsd:false
    if RequestBody → content not null **then**
        for MediaType Object in content **do**
            mediaTypeIndividual = MediaTypeTranslator.translate(MediaType)
            requestBodyIndividual $\xrightarrow{openapi2:content}$ mediaTypeIndividual

LISTING 4.11: OpenAPI Request Body object example.

```yaml
requestBody:
    description: Update an existent pet in the store
    required: true
    content:
      application/json:
        schema:
          type: object
          properties:
            id:
              type: integer
              format: int64
```

LISTING 4.12: RequestBody Class individual resulting from the translation of the OpenAPI Request Body object of Listing 4.11

```
1  % RequestBody Class individual
2
3   ex:requestBody1 a openapi2:RequestBody;
4     openapi2:description "Update an existent pet in the
          store";
5     openapi2:required xsd:true;
6     openapi2:content ex:mediaType1 .
7
8  % application/json MediaType Class individual
9
10 ex:mediaType1 a openapi2:MediaType;
11    openapi2:mediaTypeName "application/json";
12    openapi2:schema ex:mediaType1Schema_NodeShape .
13
14 % SHACL NodeShape for schema
15
16 ex:mediaType1Schema_NodeShape a sh:NodeShape;
17    rdfs:label "mediaType1SchemaNodeShape";
18    sh:targetClass ex:mediaType1Schema;
19    sh:property ex:mediaType1Schema_id_PropertyShape .
20
21 ex:mediaType1Schema a owl:Class .
22
23 % SHACL PropertyShape for id property
24
25 ex:mediaType1Schema_id_PropertyShape a sh:PropertyShape;
26    rdfs:label "mediaType1Schema_id_PropertyShape";
27    openapi2:name "id";
28    sh:path ex:mediaType1Schema_id;
29    sh:datatype xsd:long .
30
31 ex:mediaType1Schema_id a rdf:Property .
```

4.8 *Transforming OpenAPI Security Scheme Objects into OpenAPI ontology individuals*

OpenAPI supports five types of security schemes namely, HTTP authentication, an API key (in the form of header, cookie, or query parameter), mutual TLS (client certificate), OAuth2's common flows (implicit, password, client credentials, authorization code) and OpenID Connect Discovery. In the ontology, the *Security Class* describes the security schemes supported by an API. The security class is generic and it is further analyzed in five different subclasses, one for each of the different Security Scheme types supported by OpenAPI:

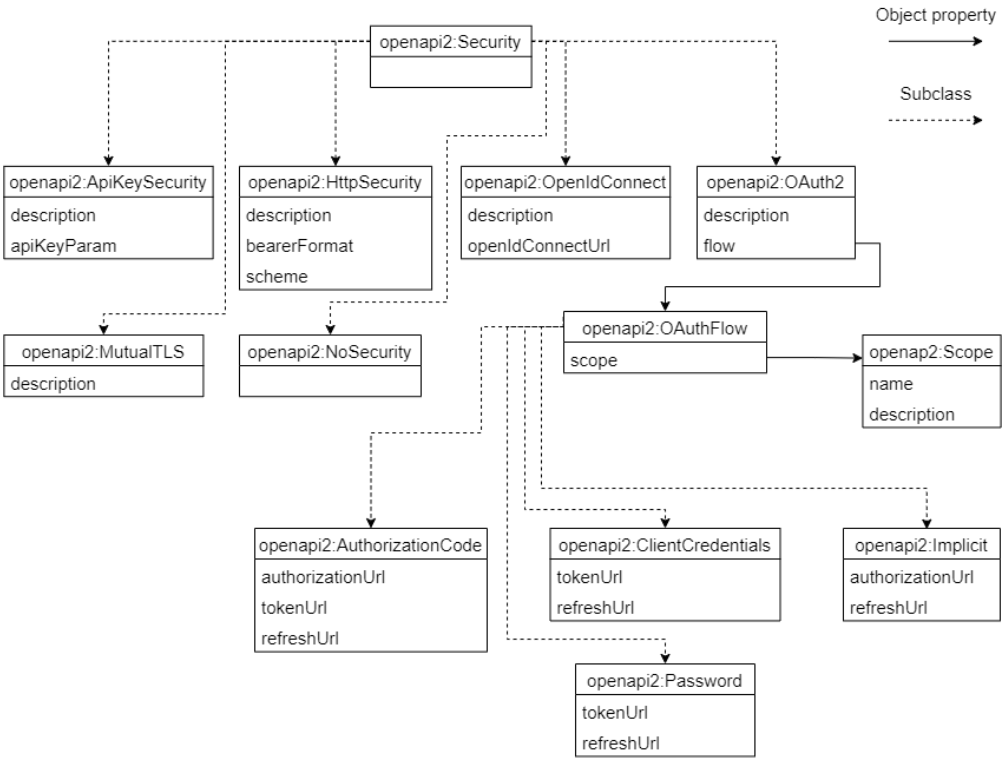
- **ApiKeySecurity Class**, describes the API key security scheme
- **HttpSecurity Class**, describes the HTTP authentication scheme
- **OAuth2 Class**, describes the OAuth2 common flows scheme
- **OpenIdConnect Class**, describes the OpenID Connect discovery scheme
- **MutualTLS Class**, describes the mutual TLS security scheme
- **NoSecurity Class**, describes the absence of security

For the OAuth2 common flows security scheme we implemented the *OAuthFlow Class*, which describes the OAuth flows supported by OAuth2. The OAuthFlow Class is generic and it is specialized in the following subclasses:

- **AuthorizationCode Class**, represents the configuration of the OAuth Authorization Code flow
- **ClientCredentials Class**, represents the configuration of the OAuth Client Credentials flow
- **Implicit Class**, represents the configuration of the OAuth Implicit flow
- **Password Class**, represents the configuration of the OAuth Resource Owner Password flow

Figure 4.2 provides the visual representation of the above classes.

FIGURE 4.2: Class Diagram of the OpenAPI Ontology Security Classes



Algorithm 9 describes the transformation of a SecurityScheme Object to one of the Security Classes. The algorithm starts by checking the *type* property of the SecurityScheme Object, which indicates what type of security scheme is described. If the type matches "APIKEY", an *ApiKeySecurity Class* individual will be initialized, and the algorithm will then check the *in* property of the SecurityScheme Object which indicates where the API key parameter is located. If the value matches "COOKIE", a Cookie Class individual will be created representing a Cookie API key parameter. Using the same logic, a QueryParameter or Header individual will be created if the API key parameter is located in "QUERY" or "HEADER" respectively. If the SecurityScheme Object has a *name* property, its value will be assigned to the apiKeyParamIndividual using the *openapi2:name* property. The above apiKeyParamIndividual will be then associated with the ApiKeySecurity individual using the *openapi2:apiKeyParam* property.

In the case of the "HTTP" type SecurityScheme object, the algorithm will initialize a *HttpSecurity Class* object. The value of *scheme* (required) property of the SecurityScheme object will be assigned to the HttpSecurity individual using the *openapi2:scheme* property. The algorithm will also check for the *bearerFormat* property value and assign it to the individual using the *openapi2:bearerFormat* property. For the "OAUTH2" case, the algorithm will initialize an *OAuth2 Class* individual. Then it will loop through the supported flows represented by Flow Objects. After

the transformation, a list of Flow individuals will be generated, which will be assigned to the OAuth2 individual using the *openapi2:flow* property.

If the SecurityScheme object is of type "OPENIDCONNECT", the algorithm will initialize a *OpenIdConnect Class* individual. It will then fetch the *openIdConnectUrl* value from the object and assign it to the individual using the *openapi2:openIdConnectUrl* property. In the last case, the type is "MUTUALTLS" and the algorithm will generate a *MutualTLS Class* individual. If a *description* property is assigned to the SecurityScheme object, then its value will be assigned to the individual using the *openapi2:description* property.

Algorithm 9 SecurityScheme Object Converter

1: **procedure** TRANSLATE(SecurityScheme secScheme)
2: **if** (secScheme → type) = "APIKEY" **then** ▷ Type is API key
3: secSchemeIndividual $\xrightarrow{rdf:type}$ openapi2:ApiKeySecurity
4: **if** (secScheme → in) = "COOKIE" **then** ▷ Initialize API key Parameter Individual
5: apiKeyParamIndividual $\xrightarrow{rdf:type}$ openapi2:Cookie
6: **else if** (secScheme → in) = "QUERY" **then**
7: apiKeyParamIndividual $\xrightarrow{rdf:type}$ openapi2:QueryParameter
8: **else if** (secScheme → in) = "HEADER" **then**
9: apiKeyParamIndividual $\xrightarrow{rdf:type}$ openapi2:Header
10: **if** (secScheme → name) not null **then**
11: apiKeyParamIndividual $\xrightarrow{openapi2:name}$ name
12: secSchemeIndividual $\xrightarrow{openapi2:apiKeyParam}$ apiKeyParamIndividual
13: **else if** (secScheme → type) = "HTTP" **then** ▷ Type is HTTP Authentication
14: secSchemeIndividual $\xrightarrow{rdf:type}$ openapi2:HttpSecurity
15: **if** (secScheme → scheme) not null **then** ▷ Scheme name is required
16: secSchemeIndividual $\xrightarrow{openapi2:scheme}$ scheme
17: **if** (secScheme → bearerFormat) not null **then** ▷ Bearer token format
18: secSchemeIndividual $\xrightarrow{openapi2:bearerFormat}$ bearerFormat

Algorithm 9 SecurityScheme Object Converter - Part 2

19: **else if** (secScheme → type) = "OAUTH2" **then** ▷ Type is OAuth2's common flows
20: secSchemeIndividual $\xrightarrow{rdf:type}$ openapi2:OAuth2

21: **if** (secScheme → flows) not null **then** ▷ Supported flow types
22: flowIndividualList = FlowTranslator(flows)

23: **for** flowIndividual in flowIndividualList **do**
24: secSchemeIndividual $\xrightarrow{openapi2:flow}$ flowIndividual

25: **else if** (secScheme → type) = "OPENIDCONNECT" **then** ▷ Type is OpenID Connect Discovery
26: secSchemeIndividual $\xrightarrow{rdf:type}$ openapi2:OpenIdConnect

27: **if** (secScheme → openIdConnectUrl) not null **then** ▷ OpenId Connect URL is required
28: secSchemeIndividual $\xrightarrow{openapi2:openIdConnectUrl}$ openIdConnectUrl

29: **else if** (secScheme → type) = "MUTUALTLS" **then** ▷ Type is mutual TLS
30: secSchemeIndividual $\xrightarrow{rdf:type}$ openapi2:MutualTLS

31: **if** (secScheme → description) not null **then** ▷ Optional description
32: secSchemeIndividual $\xrightarrow{openapi2:description}$ description

Algorithm 10 describes the transformation of an OpenAPI OAuthFlow Object into an individual of one of the four subclasses shown above based on the flow type.

Algorithm 10 OAuthFlow Object Converter

```
1: procedure TRANSLATE(OAuthFlow oauthflow)
2:     if (oauthflow → authorizationCode) not null then
3:         flowIndividual --rdf:type--> openapi2:AuthorizationCode
4:         if (oauthflow → authorizationUrl) not null then
5:             flowIndividual --openapi2:authorizationUrl--> IRI(authorizationUrl)
6:         if (oauthflow → tokenUrl) not null then
7:             flowIndividual --openapi2:tokenUrl--> IRI(tokenUrl)
8:         if (oauthflow → refreshUrl) not null then
9:             flowIndividual --openapi2:refreshUrl--> IRI(refreshUrl)
10:        if (oauthflow → scopes) not null then
11:            translateScopes(scopes, flowIndividual)
12:    else if (oauthflow → clientCredentials) not null then
13:        flowIndividual --rdf:type--> openapi2:ClientCredentials
14:        if (oauthflow → tokenUrl) not null then
15:            flowIndividual --openapi2:tokenUrl--> IRI(tokenUrl)
16:        if (oauthflow → refreshUrl) not null then
17:            flowIndividual --openapi2:refreshUrl--> IRI(refreshUrl)
18:        if (oauthflow → scopes) not null then
19:            translateScopes(scopes, flowIndividual)
20:    else if (oauthflow → implicit) not null then
21:        flowIndividual --rdf:type--> openapi2:Implicit
22:        if (oauthflow → authorizationUrl) not null then
23:            flowIndividual --openapi2:authorizationUrl--> IRI(authorizationUrl)
24:        if (oauthflow → refreshUrl) not null then
25:            flowIndividual --openapi2:refreshUrl--> IRI(refreshUrl)
26:        if (oauthflow → scopes) not null then
27:            translateScopes(scopes, flowIndividual)
```

Algorithm 10 OAuthFlow Object Converter - Part 2

28: **else if** (oauthflow → password) not null **then**
29: flowIndividual $\xrightarrow{rdf:type}$ openapi2:Password
30: **if** (oauthflow → tokenUrl) not null **then**
31: flowIndividual $\xrightarrow{openapi2:tokenUrl}$ IRI(tokenUrl)
32: **if** (oauthflow → refreshUrl) not null **then**
33: flowIndividual $\xrightarrow{openapi2:refreshUrl}$ IRI(refreshUrl)
34: **if** (oauthflow → scopes) not null **then**
35: translateScopes(scopes, flowIndividual)

36: **procedure** TRANSLATESCOPES(Scopes scopes, flowIndividual)
37: **for** Scope Object in scopes **do**
38: scopeIndividual $\xrightarrow{rdf:type}$ openapi2:Scope
39: **if** (Scope → name) not null **then**
40: scopeIndividual $\xrightarrow{openapi2:name}$ name
41: **if** (Scope → description) not null **then**
42: scopeIndividual $\xrightarrow{openapi2:description}$ description
43: flowIndividual $\xrightarrow{openapi2:scope}$ scopeIndividual ▷ Connect Flow, Scope individuals

LISTING 4.13: OpenAPI SecurityScheme object example.

```
securitySchemes:
    petstore_auth:
      type: oauth2
      flows:
        implicit:
          authorizationUrl: https://petstore3.swagger.io/
              oauth/authorize
          scopes:
            write:pets: modify pets in your account
            read:pets: read your pets
```

Listing 4.14: OAuth2 Class individual resulting from the translation of the OpenAPI SecurityScheme object of Listing 4.13

```
1  % OAuth2 Class Individual created for petstore_auth
       SecurityScheme
2
3  ex:securityScheme1 a openapi2:OAuth2;
4    openapi2:flow ex:implicitFlow1 .
5
6  % Implicit Class flow individual
7
8  ex:implicitFlow1 a openapi2:Implicit;
9    openapi2:authorizationUrl <https://petstore3.swagger.io
         /oauth/authorize>;
10   openapi2:scope ex:scope1, ex:scope2 .
11
12 % Scope Class individual created for write:pets scope
13
14 ex:scope1 a openapi2:Scope;
15   openapi2:name "write:pets";
16   openapi2:description "modify pets in your account" .
17
18 % Scope Class individual created for read:pets scope
19
20 ex:scope2 a openapi2:Scope;
21   openapi2:name "read:pets";
22   openapi2:description "read your pets" .
```

4.9 *Transforming OpenAPI Operation objects to Operation Class individuals*

An OpenAPI Operation object describes a single operation on an API path. Algorithm 11 describes the procedure for converting an OpenAPI Operation Object into an individual of the *Operation Class*. The algorithm begins by initializing an Operation Class individual and assigning the properties *openapi2:onPath* and *openapi2:method* to it (lines 2-4). The *openapi2:onPath* property takes a Path Class individual as value, which represents the API path of the Operation. The available values for the *openapi2:method* property are individuals of the *Http Method* Class. The ontology defines the following individuals for the *Http Method* Class :

- **openapi2:DELETE**
- **openapi2:GET**
- **openapi2:HEAD**
- **openapi2:OPTIONS**
- **openapi2:PATCH**
- **openapi2:POST**
- **openapi2:PUT**
- **openapi2:TRACE**

Following the initialization, the algorithm will handle the operation server selection. As stated in the OpenAPI Documentation, if there are no servers defined in the Operation object then servers defined in the PathItem object (path servers) will be used instead. Furthermore, if servers are not defined in the PathItem object, then servers defined in the OpenAPI object will be used in this operation. The selected *Server Class* will be assigned to the operation individual using the *openapi2:serverInfo* property (lines 5-15). The Operation object may have one or more Tag objects. Tag objects are also defined in the OpenAPI object. If the operation's tags come from the OpenAPI object then the algorithm will fetch the Tag Class individuals (produced from earlier translation steps) from the cache and assign them to the operation individual using the *openapi2:Tag* property (lines 18-26). If an operation tag does not exist in the OpenAPI object, then a new Tag Class individual will be created and assigned to the operation individual using the same property.

In lines 27-30 the algorithm will check for the *summary* and *description* properties of the operation object and assign their values to the operation individual using the properties *openapi2:summary* and *openapi2:description*. An ExternalDocumenation object can also be included inside an Operation object. The algorithm will convert it to an ExternalDoc Class individual and assign it to the operation individual using the *openapi2:externalDoc* property (lines 31-34). An Operation object may include a unique identifier, which is the value of the *operationId* property. We assign this value to the operation individual using the *openapi2:name* property.

For the Parameter objects included inside the Operation object, the algorithm loops through them and checks for a reference. If the Parameter object has a reference to a reusable Parameter object (in the Components section), the algorithm will fetch the individual produced by that Parameter Object from the cache. If the Parameter object is defined inline then the algorithm will convert the Parameter object to an individual of the appropriate class. If the Parameter object describes a Path Parameter, the algorithm will assign the parameter individual to the operation individual using the *openapi2:pathParam* property. If it is a Query Parameter the algorithm will do the same using the *openapi2:queryParam* property, if it is a Cookie the algorithm will use the *openapi2:cookie* property, and lastly for a Header the *openapi2:requestHeader* property will be used (lines 37-50).

A RequestBody object may be included in the Operation object. The algorithm scans the Request Body object for a reference to a reusable one in the components section. If a reference exists, then the RequestBody class individual for that object will be fetched from cache, otherwise the inline object will be converted to an individual. The RequestBody Class individual will be assigned to the operation individual using the *openapi2:requestBody* property (lines 51-56).

An Operation object holds one or more response objects which describe the outcome of the request to the API. The algorithm scans all available status codes for a Response object. If the Response object contains a reference to a reusable object in the components section, the algorithm will fetch the Response Class individual for that object, otherwise the inline object will be converted. The Response Class individual will be assigned to the operation individual using the *openapi2:response* property (lines 57-66).

For operation Callback Objects, the algorithm will loop through the different callback events and for every callback url declared it will fetch the PathItem object which contains the callback operations. Every callback Operation Object will be converted to a Callback Class individual and assigned to the operation individual using the *openapi2:callback* property (lines 67-74).

The algorithm will then scan for the *deprecated* property and will assign its value to the operation individual using the *openapi2:deprecated* property. If the *deprecated* property is not declared in the Operation object then the *openapi2:deprecated* assigned to the individual will have the default value which is false.

Finally, the algorithm will make the operation's Security Requirement object selection. The algorithm will scan for Security Requirement objects defined in the Operation object. If such objects exist, then they will be converted to *SecurityRequirement Class* individuals and will be assigned to the operation individual using the *openapi2:security* property. If Security Requirement objects are not defined in the Operation object the algorithm will try to fetch (from the cache) the *SecurityRequirement Class* individuals that were produced from the translation of the OpenAPI object's SecurityRequirements . If no Security Requirement objects are defined neither in the OpenAPI object nor in the Operation object itself, then a *NoSecurity (SecurityRequirement Class) individual* will be assigned to the operation individual using the *openapi2:security* property.

Algorithm 11 Operation Object Converter

```
1: procedure TRANSLATE(Operation op, String method, String path)
2:     opIndividual --rdf:type--> openapi2:Operation
3:     opIndividual --openapi2:onPath--> path
4:     opIndividual --openapi2:method--> method     ▷ Could be openapi2:PUT,
   openapi2:POST etc.
                                                    ▷ Priority for Servers
5:     List pathItemServerIndividuals =
   cache.getPathItemServerIndividuals(path)
6:     List rootServerIndividuals = cache.getRootServerIndividuals()
7:     List preferredServerIndividuals
8:     if op → servers not null then
9:         for so in servers do
10:            opsi = ServerTranslator.translate(so)
11:            preferredServerIndividuals.add(opsi)
12:    else if pathItemServerIndividuals.size() > 0 then
13:        preferredServerIndividuals = pathItemServerIndividuals
14:    else if rootServerIndividuals.size() > 0 then
15:        preferredServerIndividuals = rootServerIndividuals
16:    for serverIndivdual in preferredServerIndividuals do
17:        opIndividual --openapi2:serverInfo--> serverIndivdual
18:    if op → tags not null then
19:        for tagName in tags do
20:            cachedTagIndividual = cache.getTagIndividual(tagName)
21:            if cachedTagIndividual != null then
22:                opIndividual --openapi2:Tag--> cachedTagIndividual
23:            else
24:                tagIndividual --rdf:type--> openapi2:Tag
25:                tagIndividual --openapi2:name--> tagName
26:                opIndividual --openapi2:Tag--> tagIndividual
```

Algorithm 11 Operation Object Converter - Part2

27: **if** op → summary not null **then**
28: opIndividual $\xrightarrow{openapi2:summary}$ summary

29: **if** op → description not null **then**
30: opIndividual $\xrightarrow{openapi2:description}$ description

31: **if** op → externalDocs not null **then**
32: **for** extDoc in externalDocs **do**
33: extDocIndividual = ExternalDocTranslator(extDoc)
34: opIndividual $\xrightarrow{openapi2:externalDoc}$ extDocIndividual

35: **if** op → operationId **then**
36: opIndividual $\xrightarrow{openapi2:name}$ operationId

37: **if** op → parameters not null **then**
38: **for** param in parameters **do**
39: **if** param → $ref not null **then**
40: parameterIndividual = cache.getParameterIndividual($ref)
41: **else**
42: parameterIndividual = ParameterTranslator(param)
43: **if** param → in = "path" **then**
44: opIndividual $\xrightarrow{openapi2:pathParam}$ parameterIndividual
45: **else if** param → in = "query" **then**
46: opIndividual $\xrightarrow{openapi2:queryParam}$ parameterIndividual
47: **else if** param → in = "cookie" **then**
48: opIndividual $\xrightarrow{openapi2:cookie}$ parameterIndividual
49: **else if** param → in = "header" **then**
50: opIndividual $\xrightarrow{openapi2:requestHeader}$ parameterIndividual

51: **if** op → requestBody not null **then**
52: **if** requestBody → $ref not null **then**
53: requestBodyIndividual = cache.getRequestBodyIndividual($ref)
54: **else**
55: requestBodyIndividual = RequestBodyTranslator(requestBody)
56: opIndividual $\xrightarrow{openapi2:requestBody}$ requestBodyIndividual

Algorithm 11 Operation Object Converter - Part3

```
57:  if op → responses not null then
58:      for statusCode in responses.keys() do
59:          apiResponse = responses.get(statusCode)
60:          if apiResponse → $ref not null then
61:              refApiResponse = fetchResponse($ref)
62:              refApiResponseIndividual = ResponseTranslator(refApiResponse, statusCode)
63:              opIndividual --openapi2:response--> refApiResponseIndividual
64:          else
65:              responseIndividual = ResponseTranslator(apiResponse, statusCode)
66:              opIndividual --openapi2:response--> responseIndividual

67:  if op → callbacks not null then
68:      for eventName in callbacks do
69:          callback = callbacks.get(eventName)
70:          for callbackUrl in callback do
71:              callbackPathItem = callback.get(callbackUrl)
72:              for callbackOp in callbackPathItem do
73:                  callbackIndividual = WebhookTranslator.translate(callbackOp)
74:                  opIndividual --openapi2:callback--> callbackIndividual

75:  if op → deprecated not null then
76:      if deprecated = true then
77:          opIndividual --openapi2:deprecated--> xsd:true
78:  else
79:      opIndividual --openapi2:deprecated--> xsd:false

80:  rootSecReqIndividual = cache.getRootSecurityRequirementIndividual()
81:  noSecReqIndividual = cache.getNoSecurityRequirementIndividual()
82:  if op → security not null then
83:      secReqIndividual = SecurityRequirementTranslator.translate(security)
84:      opIndividual --openapi2:security--> secReqIndividual
85:  else if rootSecReqIndividual not null then
86:      opIndividual --openapi2:security--> rootSecReqIndividual
87:  else if noSecReqIndividual not null then
88:      opIndividual --openapi2:security--> noSecReqIndividual
```

Listing 4.15 contains a simple OpenAPI Operation object. This object describes a GET operation in the path /user/login. The individuals resulting from the translation are shown in Listing 4.16.

Listing 4.15: OpenAPI Operation object example.

```
paths:
  /user/login:
    get:
      tags:
        - user
      summary: Logs user into the system
      operationId: loginUser
      parameters:
        - name: username
          in: query
          description: The user name for login
          required: false
          schema:
            type: string
        - name: password
          in: query
          description: The password for login in clear
              text
          required: false
          schema:
            type: string
      responses:
        '400':
          description: Invalid username/password supplied
```

Listing 4.16: Operation Class individual resulting from the translation of the OpenAPI Operation object of Listing 4.15

```
ex:tag3 a openapi2:Tag;
   openapi2:name "user";
   openapi2:description "Operations about user" .

ex:server1 a openapi2:Server;
   openapi2:host "https://petstore3.swagger.io/api/v3" .

ex:path1 a openapi2:Path;
   openapi2:pathName "/user/login" .

ex:op1 a openapi2:Operation;
   openapi2:onPath ex:path1;
   openapi2:method openapi2:GET;
   openapi2:serverInfo ex:server1;
   openapi2:tag ex:tag3;
   openapi2:summary "Logs user into the system";
   openapi2:description "";
   openapi2:name "loginUser";
   openapi2:queryParam ex:param1, ex:param2;
   openapi2:response ex:response1;
   openapi2:deprecated xsd:false;
   openapi2:security ex:securityRequirement1 .

ex:noSecurity a openapi2:NoSecurity .

ex:securityRequirement1 a openapi2:SecurityRequirmentItem;
   openapi2:securityItem ex:noSecurity .

ex:param1 a openapi2:QueryParameter;
   openapi2:name "username";
   openapi2:descriptjon "The user name for login";
   openapi2:required xsd:false;
   openapi2:style openapi2:form;
   openapi2:explode xsd:true;
   openapi2:schema ex:param1Schema_PropertyShape;
   openapi2:allowEmptyValue xsd:false;
   openapi2:allowReserved xsd:false;
   openapi2:deprecated xsd:false .

ex:param1Schema_PropertyShape a sh:PropertyShape;
   rdfs:label "param1Schema_PropertyShape";
   openapi2:name "param1Schema";
   sh:path ex:param1Schema;
   sh:datatype xsd:string .

ex:param1Schema a rdf:Property .

ex:param2 a openapi2:QueryParameter;
   openapi2:name "password";
   openapi2:descriptjon "The password for login in clear
      text";
   openapi2:required xsd:false;
   openapi2:style openapi2:form;
   openapi2:explode xsd:true;
   openapi2:schema ex:param2Schema_PropertyShape;
   openapi2:allowEmptyValue xsd:false;
   openapi2:allowReserved xsd:false;
   openapi2:deprecated xsd:false .

ex:param2Schema_PropertyShape a sh:PropertyShape;
   rdfs:label "param2Schema_PropertyShape";
   openapi2:name "param2Schema";
   sh:path ex:param2Schema;
   sh:datatype xsd:string .

ex:param2Schema a rdf:Property .

ex:response1 a openapi2:4xxResponse;
   openapi2:statusCode "400"^^xsd:int;
   openapi2:description "Invalid username/password
      supplied" .
```

Chapter 5

Transforming Webhooks and Operation Callbacks to Webhook Class individuals

Webhooks are asynchronous server requests initiated other than by an API call, to a client specific endpoint, for example by an out of band registration. In OpenAPI 3.1 webhooks are defined under the *webhooks* field of the OpenAPI Object. In OpenAPI a webhook consists of a unique name and a *PathItem* Object which describes the requests that will be sent by the API provider whenever a certain event happens.

Operation Callbacks are almost identical to Webhooks. The main difference is that Callbacks are asynchronous server-initiated requests to a client specific endpoint when a specific event occurs (an API call for example). Because of their similarity we describe them using the same Webhook Class in the ontology.

Algorithm 12 analyzes the translation of an OpenAPI Webhook into a *Webhook Class* individual. In lines 2-3 a *Webhook Class* individual is initialized then the *openapi2:method* property is assigned to it using the Webhook's request method as value (lines 2-3). In lines 4-12 the algorithm loops through the optional tags contained inside the Webhook's Operation object and assigns their values using the *openapi2:tag* property to the *Webhook Class* individual. Optional summary, description and external documentation is handled by the algorithm in lines 13-22. Every webhook comes with unique event name which declares the event that triggers the webhook. The event name value is assigned to the *Webhook Class* individual using the *openapi2:name* property. The Webhook Operation object also contains optional parameters. In lines 25-43 the algorithm processes each Parameter object. The Parameter object may contain a reference to a reusable Parameter object in the OpenAPI Components section or it can also be defined inline. If the Parameter object has a reference the algorithm fetches the individual produced from the translation of that Parameter object earlier from cache, otherwise (defined inline) it gets converted using the ParameterTranslator module. Based on the *in* property of the Parameter object the according property will be used to assign the

Parameter individual to the *Webhook Class* individual as seen in lines 35-43.

Algorithm 12 Webhook Object Converter

1: **procedure** TRANSLATE(Operation op, String method, String eventName)
2: webhookIndividual $\xrightarrow{rdf:type}$ openapi2:Webhook
3: webhookIndividual $\xrightarrow{openapi2:method}$ method

4: **for** tag in op → tags **do**
5: cacheTagIndividual = cache.getTagIndividual(tag)
6: **if** cacheTagIndividual not null **then**
7: WebhookIndividual $\xrightarrow{openapi2:tag}$ cacheTagIndividual
8: **else**
9: tagIndividual $\xrightarrow{rdf:type}$ openapi2:Tag
10: tagIndividual $\xrightarrow{openapi2:name}$ tag
11: webhookIndividual $\xrightarrow{openapi2:tag}$ tagIndividual
12: cache.addTagIndividual(tag, tagIndividual)

13: webhookSummary = op → summary
14: **if** webhookSummary not null **then**
15: webhookIndiviudal $\xrightarrow{openapi2:summary}$ webhookSummary

16: webhookDescription = op → description
17: **if** webhookDescription not null **then**
18: webhookIndiviudal $\xrightarrow{openapi2:description}$ webhookDescription

19: webhookExternalDoc = op → externalDocs
20: **if** webhookExternalDoc not null **then**
21: externalDocIndividual = ExternalDocTranslator.translate(webhookExternalDoc)
22: webhookIndiviudal $\xrightarrow{openapi2:externalDoc}$ externalDocIndividual

23: **if** eventName not null **then**
24: webhookIndiviudal $\xrightarrow{openapi2:name}$ eventName

Algorithm 12 Webhook Object Converter - Part 2

```
25: if op → parameters not null then
26:     for p in parameters do
27:         ref = p → $ref
28:         if ref not null then
29:             refParameter = fetchParameter(ref)
30:             targetParameterIn = refParameter → in
31:             targetIndividual = cache.getRefParameterIndividual(ref)
32:         else
33:             targetParameterIn = p → in
34:             targetIndividual = ParameterTranslator.translate(p)
35:         if targetParameterIn == "path" then
36:             targetProperty = "openapi2:pathParam"
37:         else if targetParameterIn == "query" then
38:             targetProperty = "openapi2:queryParam"
39:         else if targetParameterIn == "cookie" then
40:             vtargetProperty = "openapi2:cookie"
41:         else if targetParameterIn == "header" then
42:             targetProperty = "openapi2:requestHeader"

43:         webhookIndividual --targetProperty--> targetIndividual

44: if o → requestBody not null then
45:     ref = requestBody → $ref
46:     if ref not null then
47:         cacheIndividual = cache.getRequestBodyRefIndividual(ref)
48:         if cacheIndividual not null then
49:             webhookIndividual --openapi2:requestBody--> cacheIndividual
50:     else
51:         requestBodyIndividual = RequestBodyTranslator.translate(requestBody)
52:         webhookIndividual --openapi2:requestBody--> requestBodyIndividual

53: if o → responses not null then
54:     for apiResponse in responses do
55:         ref = apiResponse → ref
56:         if ref not null then
57:             targetResponse = fetchResponse(ref)
58:         else
59:             targetResponse = apiResponse
60:         responseIndividual = ResponseTranslator.translate(targetResponse, statusCode)
61:         webhookIndividual --openapi2:response--> responseIndividual

62: deprec = o → deprecated
63: if deprec not null AND deprec = true then
64:     webhookIndividual --openapi2:deprecated--> xsd:true
65: else
66:     webhookIndividual --openapi2:deprecated--> xsd:false
```

Listing 5.1 contains an Operation Callback example. The "POST \subscribe" operation contains a Callback which is a server-initiated POST request to the endpoint defined by "callbackUrl" of request body. Also contains a 200 response which is the response the client API must send to the server if it accepts the callback.

Listing 5.1: OpenAPI Operation Callback example.

```
1  paths:
2    /subscribe:
3      post:
4        summary: Subscribe to a webhook
5        callbacks:
6          myEvent:
7            '{$request.body#/callbackUrl}':
8              post:
9                requestBody:
10                 required: true
11                 content:
12                   application/json:
13                     schema:
14                       type: object
15                       properties:
16                         message:
17                           type: string
18               responses:
19                 '200':
20                   description: Your server returns this
                         code if it accepts the callback
```

In Listing 5.2 we can see the ontology individuals produced from the provided operation. In lines 1-8 the Operation individual is initialized for the "POST \subscribe" operation. The Operation Class individual also has the *openapi2:callback* property which has a Webhook Class individual as value. The Webhook Class individual (lines 16-21) represents the Callback.

LISTING 5.2: Ontology individuals resulting from the translation of the OpenAPI Operation object of Listing 5.1

```
ex:op1 a openapi2:Operation;
  openapi2:onPath ex:path1;
  openapi2:method openapi2:POST;
  openapi2:serverInfo ex:server1;
  openapi2:summary "Subscribe to a webhook";
  openapi2:callback ex:callback1;
  openapi2:deprecated xsd:false;
  openapi2:security ex:securityRequirement1 .

ex:path1 a openapi2:Path;
  openapi2:pathName "/subscribe" .

ex:server1 a openapi2:Server;
  openapi2:host "/" .

ex:callback1 a openapi2:Webhook;
  openapi2:method openapi2:POST;
  openapi2:name "myEvent";
  openapi2:requestBody ex:requestBody1;
  openapi2:response ex:response1;
  openapi2:deprecated xsd:false .

ex:response1 a openapi2:2xxResponse;
  openapi2:statusCode "200"^^xsd:int;
  openapi2:description "Your server returns this code if
     it accepts the callback" .

ex:requestBody1 a openapi2:RequestBody;
  openapi2:required xsd:true;
  openapi2:content ex:mediaType1 .

ex:mediaType1 a openapi2:MediaType;
  openapi2:mediaTypeName "application/json";
  openapi2:schema ex:mediaType1Schema_NodeShape .

ex:mediaType1Schema_NodeShape a sh:NodeShape;
  rdfs:label "mediaType1SchemaNodeShape";
  sh:targetClass ex:mediaType1Schema;
  sh:property ex:mediaType1Schema_message_PropertyShape .

ex:mediaType1Schema a owl:Class .

ex:mediaType1Schema_message_PropertyShape a sh:
    PropertyShape;
  rdfs:label "mediaType1Schema_message_PropertyShape";
  openapi2:name "message";
  sh:path ex:mediaType1Schema_message;
  sh:datatype xsd:string .

ex:mediaType1Schema_message a rdf:Property .

ex:noSecurity a openapi2:NoSecurity .

ex:securityRequirement1 a openapi2:SecurityRequirmentItem
    ;
  openapi2:securityItem ex:noSecurity .
```

Chapter 6

Transforming Schema Objects to OpenAPI Ontology individuals

In this section we are going to describe the translation procedure of an OpenAPI Schema Object to ontology components. A Schema Object allows the definition of input and output data types. In our ontology we describe Schema Objects using the SHACL [11] language.

We will start by analyzing each case of Schema Object data type. Data types supported by OAS are based on the JSON Schema Specification. Those data types are:

- **object**
- **string**
- **number**
- **integer**
- **array**
- **boolean**

There are three categories resulting from the above data types, that we are going to analyze individually: (a) The ***Primitive*** data types which are *string, number, integer, boolean*; (b) The ***Object*** data type; and (c) The ***Array*** data type.

The ***Primitive*** data types often have an optional modifier called format. An example would be a schema object with *integer* data type and *int64* format which specifies that the expected value should be a long number. In order to describe the ***Primitive*** data type-format combinations in our ontology, we used XSD data type individuals. The mapping between the supported OAS ***Primitive*** data types and XSD data types we used are presented in Table 6.1

Dataype Mapping		
OAS Datatype	Format	XSD Datatype
integer	-	integer
integer	int32	int
integer	int64	long
string	-	string
string	date	date
string	date-time	datetime
string	byte	base64binary
boolean	-	boolean
number	-	XONE(float, double, decimal, integer)
number	float	float
number	double	double

Table 6.1: OAS Primitive data type to XSD data type mapping

6.1 *Transforming Schema Objects of Primitive data type*

In this section we describe the translation of Schema Objects with Primitive data type. An example of such a Schema Object is available in Listing 6.1. The example contains an Age Schema Object with *integer* data type and *int32* format.

Listing 6.1: Primitive data type Schema Object example.

```
1 Age:
2   type: integer
3   format: int32
```

In order to describe this type of Schema Objects in our ontology, we use SHACL's *PropertyShape* construct. The ontology individuals resulting from the translation of the Schema Object of Listing 6.1 (including PropertyShape) are presented in Listing 6.2. In line 1 we create the PropertyShape individual called *Age_PropertyShape* and we set the rdf:type to sh:PropertyShape. In line 2 we provide a label for that PropertyShape, which is a human-readable version of the resource's name. In line 7 we create the Age Property individual, then we link the PropertyShape with this property individual using the *sh:path* property (as seen line line 4). Finally, we set the data type to PropertyShape using the *sh:datatype* property. The data type value that will be used based on the mapping of Table 6.1 is XSD's *int*.

Listing 6.2: Ontology individuals resulting from the translation of the Schema Object of Listing 6.1

```
1 ex:Age_PropertyShape a sh:PropertyShape;
2   rdfs:label "Age_PropertyShape";
3   openapi2:name "Age";
4   sh:path ex:Age;
5   sh:datatype xsd:int .
6
7 ex:Age a rdf:Property .
```

We can transform any ***Primitive*** data type Schema Object using the logic of the above example.

6.2 *Transforming Schema Objects of Object data type*

In this section we describe the translation of Schema Objects with *Object* data type. An example of such a Schema Object is provided in Listing 6.3. The example contains a Pet Schema Object of *Object* data type. This object contains two additional property Schema Objects, name and age, with *Primitive* data types. These schema objects are translated in a way similar to the one described in Section 6.1

Listing 6.3: OpenAPI Operation object example.

```
1 Pet:
2   type: object
3   properties:
4     name:
5       type: string
6     age:
7       type: integer
8       format: int32
```

In order to describe this type of Schema Objects in our ontology we use SHACL's NodeShape construct. The ontology individuals resulting from the translation of the Schema Object of Listing 6.3 are presented in Listing 6.4. In lines 1-4 the Pet_NodeShape is created for the Pet Schema Object. We also create a class for the Pet Object as seen in line 6, which is then linked to the NodeShape using the *sh:targetClass* property. The two properties (name,age) contained in the Pet Schema Object are *Primitive* data type Schema Objects. The resulting PropertyShape individuals for the two properties name, age are located in lines 8-14 and 16-22 respectively and they are translated using the logic described in section 6.1. The PropertyShape individuals are then linked to the owner NodeShape individual using the *sh:property* property in line 4.

Listing 6.4: OpenAPI Individuals produced by the translation of the Schema Objects from Listing 6.3

```
ex:Pet_NodeShape a sh:NodeShape;
  rdfs:label "PetNodeShape";
  sh:targetClass ex:Pet;
  sh:property ex:Pet_name_PropertyShape, ex:
     Pet_age_PropertyShape .

ex:Pet a owl:Class .

ex:Pet_name_PropertyShape a sh:PropertyShape;
  rdfs:label "Pet_name_PropertyShape";
  openapi2:name "name";
  sh:path ex:Pet_name;
  sh:datatype xsd:string .

ex:Pet_name a rdf:Property .

ex:Pet_age_PropertyShape a sh:PropertyShape;
  rdfs:label "Pet_age_PropertyShape";
  openapi2:name "age";
  sh:path ex:Pet_age;
  sh:datatype xsd:int .

ex:Pet_age a rdf:Property .
```

6.3 *Handling references inside Schema Objects*

In this section we describe the way we handle references inside Schema Objects. A developer can create a snippet element (in our case a Schema Object) which then can be reused multiple times across other API resources using references. This method saves a lot of time for the developer and makes the API Specification more compact and therefore easier to read and modify. The example in Listing 6.5 contains two Schema Objects, Pet and Age. The Pet Schema Object contains an age property. Instead of writing a new Schema Object for the age property, the developer has preferred to use a reference to the Age Schema Object which is already declared bellow.

Listing 6.5: Using $ref inside Schema Objects example.

```
1  Pet:
2    type: object
3    properties:
4      name:
5        type: string
6      age:
7        $ref: '#/components/schemas/Age'
8  Age:
9    type: integer
10   format: int32
```

In the previous sections we discussed how Schema Objects of Object data type and Primitive data type are translated. In this case, as seen in the example of Listing 6.6, we create a Property Shape called Pet_age_PropertyShape for the age property which is linked with the reference's Age_PropertyShape (created from the Age Schema Object) using the *sh:node* property. With the help of the *sh:node* property we declare that every value for the Pet_age_PropertyShape must fulfill the constraints expressed by the Age_PropertyShape. Using this method we can reuse the Age_PropertyShape instead of copying it to an identical Property Shape. Finally, if there is a reference to a previously translated Schema Object, we are using a caching mechanism to retrieve that individual and reuse it.

Listing 6.6: Individuals produced by the translation of the Schema Objects from Listing 6.5

```
ex:Pet_NodeShape a sh:NodeShape;
  rdfs:label "PetNodeShape";
  sh:targetClass ex:Pet;
  sh:property ex:Pet_name_PropertyShape, ex:
     Pet_age_PropertyShape .

ex:Pet a owl:Class .

ex:Pet_name_PropertyShape a sh:PropertyShape;
  rdfs:label "Pet_name_PropertyShape";
  openapi2:name "name";
  sh:path ex:Pet_name;
  sh:datatype xsd:string .

ex:Pet_name a rdf:Property .

ex:Age_PropertyShape a sh:PropertyShape;
  rdfs:label "Age_PropertyShape";
  openapi2:name "Age";
  sh:path ex:Age;
  sh:datatype xsd:int .

ex:Age a rdf:Property .

ex:Pet_age_PropertyShape a sh:PropertyShape;
  sh:node ex:Age_PropertyShape .
```

6.4 *Handling Schema Object keywords*

Starting from OpenAPI 3.0, keywords are allowed in Schema Objects, and they are used in order to combine schemas. Those keywords are :

- **oneOf**, data given should be valid against one of the specified schemas.
- **anyOf**, data given should be valid against any of the specified schemas.
- **allOf**, data given should be valid against all of the specified schemas.
- **not**, data given should not be valid against the specified schema.

6.4.1 *Handling the oneOf keyword*

The example of Listing 6.7 contains three different Schema Objects Dog, Cat and Pet. The Pet Schema Object contains the *oneOf* keyword along with references to the Dog and Cat Schema Objects. This indicates that data are valid against the Pet Schema Object if only they are valid against strictly one of the Dog or Cat Schema Objects.

Listing 6.7: Use of oneOf keyword example.

```
1  Dog:
2    type: object
3    properties:
4      bark:
5        type: boolean
6      breed:
7        type: string
8  Cat:
9    type: object
10   properties:
11     hunts:
12       type: boolean
13     age:
14       type: integer
15 Pet:
16   oneOf:
17     - $ref: '#/components/schemas/Dog'
18     - $ref: '#/components/schemas/Cat'
```

In the resulting ontology individuals shown in Listing 6.8, lines 1-22 contain the individuals associated with the Dog Schema Object and lines 24-45 contain the individuals associated with the Cat Schema Object. The individuals produced from the Pet Schema Object are contained in lines 47-52. In order to have the same meaning as the *oneOf* keyword in our ontology, we use the *sh:xone* (SHACL) property inside the Pet_NodeShape. The objects of that property are the NodeShapes produced from the Dog and Cat Schema Objects as seen in line 50.

Listing 6.8: Individuals produced from the translation of the Schema Objects from Listing 6.7

```
ex:Dog_NodeShape a sh:NodeShape;
  rdfs:label "DogNodeShape";
  sh:targetClass ex:Dog;
  sh:property ex:Dog_bark_PropertyShape, ex:
      Dog_breed_PropertyShape .

ex:Dog a owl:Class .

ex:Dog_bark_PropertyShape a sh:PropertyShape;
  rdfs:label "Dog_bark_PropertyShape";
  openapi2:name "bark";
  sh:path ex:Dog_bark;
  sh:datatype xsd:boolean .

ex:Dog_bark a rdf:Property .

ex:Dog_breed_PropertyShape a sh:PropertyShape;
  rdfs:label "Dog_breed_PropertyShape";
  openapi2:name "breed";
  sh:path ex:Dog_breed;
  sh:datatype xsd:string .

ex:Dog_breed a rdf:Property .

ex:Cat_NodeShape a sh:NodeShape;
  rdfs:label "CatNodeShape";
  sh:targetClass ex:Cat;
  sh:property ex:Cat_hunts_PropertyShape, ex:
      Cat_age_PropertyShape .

ex:Cat a owl:Class .

ex:Cat_hunts_PropertyShape a sh:PropertyShape;
  rdfs:label "Cat_hunts_PropertyShape";
  openapi2:name "hunts";
  sh:path ex:Cat_hunts;
  sh:datatype xsd:boolean .

ex:Cat_hunts a rdf:Property .

ex:Cat_age_PropertyShape a sh:PropertyShape;
  rdfs:label "Cat_age_PropertyShape";
  openapi2:name "age";
  sh:path ex:Cat_age;
  sh:datatype xsd:integer .

ex:Cat_age a rdf:Property .

ex:Pet_NodeShape a sh:NodeShape;
  rdfs:label "PetNodeShape";
  sh:targetClass ex:Pet;
  sh:xone (ex:Dog_NodeShape ex:Cat_NodeShape) .

ex:Pet a owl:Class .
```

6.4.2 *Handling the anyOf keyword*

The example of Listing 6.9 contains three Schema Objects, PetByAge, PetByType and Pets. The Pets Schema Object contains the *anyOf* keyword along with references to the PetByType and PetByAge Schema Objects. This aims to indicate that data against the Pets Schema Object are only valid when they are valid against at least one of the PetByType and PetByAge Schema Objects.

Listing 6.9: Use of anyOf keyword example.

```
1  PetByAge:
2    type: object
3    properties:
4      age:
5        type: integer
6      nickname:
7        type: string
8  PetByType:
9    type: object
10   properties:
11     pet_type:
12       type: string
13     hunts:
14       type: boolean
15 Pets:
16   anyOf:
17     - $ref: '#/components/schemas/PetByType'
18     - $ref: '#/components/schemas/PetByAge'
```

In the resulting ontology individuals shown in Listing 6.10, lines 1-22 contain the individuals associated with the PetByAge Schema Object and lines 24-45 contain the individuals associated with the PetByType Schema Object. The individuals produced from the Pets Schema Object are contained in lines 47-52. In order to have the same meaning as the *anyOf* keyword in our ontology, we use the *sh:or* (SHACL) property inside the Pets_NodeShape. The objects of that property are the NodeShapes produced from PetByAge and PetByType Schema Objects as seen in line 50.

Listing 6.10: Individuals produced by the translation of the Schema Objects from Listing 6.9

```
ex:PetByAge_NodeShape a sh:NodeShape;
  rdfs:label "PetByAgeNodeShape";
  sh:targetClass ex:PetByAge;
  sh:property ex:PetByAge_age_PropertyShape, ex:
      PetByAge_nickname_PropertyShape .

ex:PetByAge a owl:Class .

ex:PetByAge_age_PropertyShape a sh:PropertyShape;
  rdfs:label "PetByAge_age_PropertyShape";
  openapi2:name "age";
  sh:path ex:PetByAge_age;
  sh:datatype xsd:integer .

ex:PetByAge_age a rdf:Property .

ex:PetByAge_nickname_PropertyShape a sh:PropertyShape;
  rdfs:label "PetByAge_nickname_PropertyShape";
  openapi2:name "nickname";
  sh:path ex:PetByAge_nickname;
  sh:datatype xsd:string .

ex:PetByAge_nickname a rdf:Property .

ex:PetByType_NodeShape a sh:NodeShape;
  rdfs:label "PetByTypeNodeShape";
  sh:targetClass ex:PetByType;
  sh:property ex:PetByType_pet_type_PropertyShape, ex:
      PetByType_hunts_PropertyShape .

ex:PetByType a owl:Class .

ex:PetByType_pet_type_PropertyShape a sh:PropertyShape;
  rdfs:label "PetByType_pet_type_PropertyShape";
  openapi2:name "pet_type";
  sh:path ex:PetByType_pet_type;
  sh:datatype xsd:string .

ex:PetByType_pet_type a rdf:Property .

ex:PetByType_hunts_PropertyShape a sh:PropertyShape;
  rdfs:label "PetByType_hunts_PropertyShape";
  openapi2:name "hunts";
  sh:path ex:PetByType_hunts;
  sh:datatype xsd:boolean .

ex:PetByType_hunts a rdf:Property .

ex:Pets_NodeShape a sh:NodeShape;
  rdfs:label "PetsNodeShape";
  sh:targetClass ex:Pets;
  sh:or (ex:PetByType_NodeShape ex:PetByAge_NodeShape) .

ex:Pets a owl:Class .
```

6.4.3 *Handling the allOf keyword*

The example of Listing 6.11 contains two Schema Objects, ErrorModel and ExtendedErrorModel. The ExtendedErrorModel Schema Object contains the *allOf* keyword along with a reference to the ErrorModel Schema Object. The keyword indicates that ExtendedErrorModel is an extension of the ErrorModel Schema Object with the additional property rootCause.

LISTING 6.11: Use of allOf keyword example.

```
1  ErrorModel:
2    type: object
3    properties:
4      message:
5        type: string
6      code:
7        type: integer
8        minimum: 100
9        maximum: 600
10 ExtendedErrorModel:
11   allOf:
12     - $ref: '#/components/schemas/ErrorModel'
13     - type: object
14       properties:
15         rootCause:
16           type: string
```

In the resulting ontology individuals shown in Listing 6.12, lines 1-22 contain the individuals associated with the ErrorModel Schema Object and lines 24-38 contain the individuals associated with the ExtendedErrorModel Schema Object. As said earlier, the ExtendedErrorModel Schema object is an extension of the ErrorModel Schema Object by using the *allOf* keyword. In order to have the same meaning as the *allOf* keyword in our ontology, we use the *sh:and* (SHACL) property inside the ExtendedErrorModel_NodeShape. The objects of this property are the ErrorModel_NodeShape and the PropertyShape produced from the additional property rootCause as seen in line 27.

Listing 6.12: Individuals produced by the translation of the Schema Objects from Listing 6.11

```
ex:ErrorModel_NodeShape a sh:NodeShape;
  rdfs:label "ErrorModelNodeShape";
  sh:targetClass ex:ErrorModel;
  sh:property ex:ErrorModel_message_PropertyShape, ex:
     ErrorModel_code_PropertyShape .

ex:ErrorModel a owl:Class .

ex:ErrorModel_message_PropertyShape a sh:PropertyShape;
  rdfs:label "ErrorModel_message_PropertyShape";
  openapi2:name "message";
  sh:path ex:ErrorModel_message;
  sh:datatype xsd:string .

ex:ErrorModel_message a rdf:Property .

ex:ErrorModel_code_PropertyShape a sh:PropertyShape;
  rdfs:label "ErrorModel_code_PropertyShape";
  openapi2:name "code";
  sh:path ex:ErrorModel_code;
  sh:datatype xsd:integer .

ex:ErrorModel_code a rdf:Property .

ex:ExtendedErrorModel_NodeShape a sh:NodeShape;
  rdfs:label "ExtendedErrorModelNodeShape";
  sh:targetClass ex:ExtendedErrorModel;
   sh:and (ex:ErrorModel_NodeShape
       ex:ExtendedErrorModel_AllOfInline_1_rootCause_PropertyShape)
       .

ex:ExtendedErrorModel a owl:Class;
  rdfs:subClassOf ex:ErrorModel .

ex:ExtendedErrorModel_AllOfInline_1_rootCause_PropertyShape a sh:
   PropertyShape;
  rdfs:label "
     ExtendedErrorModel_AllOfInline_1_rootCause_PropertyShape
     ";
  openapi2:name "
     ExtendedErrorModel_AllOfInline_1_rootCause";
  sh:path ex:ExtendedErrorModel_AllOfInline_1_rootCause;
  sh:datatype xsd:string .

ex:ExtendedErrorModel_AllOfInline_1_rootCause a rdf:
   Property .
```

6.4.4 *Handling the not keyword*

The example of Listing 6.13 contains a PetByType Schema Object. The pet_type property of this Schema Object contains the *not* keyword which indicates that the data of this property should not be of type integer.

Listing 6.13: Use of allOf keyword example.

```
1 PetByType:
2   type: object
3   properties:
4     pet_type:
5       not:
6         type: integer
```

The resulting ontology individuals are shown in Listing 6.14. Lines 1-14 contain the NodeShape produced from the PetByType Schema Object and the PropertyShape produced from the pet_type property. In order to represent the *not* keyword inside our ontology we use the *sh:not* (SHACL) property. The object of this property is the PropertyShape produced from the proprety contained inside the *not* keyword as seen in lines 16-19.

Listing 6.14: Individuals produced by the translation of the Schema Objects from Listing 6.13

```
1  ex:PetByType_NodeShape a sh:NodeShape;
2    rdfs:label "PetByTypeNodeShape";
3    sh:targetClass ex:PetByType;
4    sh:property ex:PetByType_pet_type_PropertyShape .
5
6  ex:PetByType a owl:Class .
7
8  ex:PetByType_pet_type_PropertyShape a sh:PropertyShape;
9    rdfs:label "PetByType_pet_type_PropertyShape";
10   openapi2:name "pet_type";
11   sh:path ex:PetByType_pet_type;
12   sh:not (ex:PetByType_pet_type_Not_PropertyShape) .
13
14 ex:PetByType_pet_type a rdf:Property .
15
16 ex:PetByType_pet_type_Not_PropertyShape a sh:PropertyShape;
17   rdfs:label "PetByType_pet_type_Not_PropertyShape";
18   openapi2:name "pet_type_Not";
19   sh:datatype xsd:integer .
```

6.5 *Handling Semantic Annotations inside Schema Objects*

A human reader of an OpenAPI specification file can easily understand the semantics of various elements by their names, descriptions or other information. However a machine needs an explicit declaration of the element semantics. The following semantic annotations were introduced in an older work [15] to provide this information to the machine:

- **x-refersTo**, the concept in a semantic model that describes an OAS element.
- **x-kindOf**, a specilization between an OAS element and a concept in a semantic model.
- **x-mapsTo**, an OAS element which is semantically similar with another OAS element.
- **x-collectionOn**, a model describes a collection over a specific property.

The value of ***x-refersTo*** and ***x-kindOf*** is the URI of a concept in a semantic model. The value of ***x-mapsTo*** is a reference to a Schema Object in the OpenAPI description. Finally, the value ***x-collectionOn*** is the name of the Schema Object's property that is an array holding a collection of items.

6.5.1 Handling the x-refersTo semantic annotation property

The x-refersTo property is responsible for associating OpenAPI elements and concepts in a semantic model. Listing 6.15 presents the usage of x-refersTo extension inside a Pet Schema Object. In this particular example, the extension associates the Pet Schema Object with a "Pet" Class and the id property with an "Id" Class, both declared in the "https://example.com/ontology" domain.

Listing 6.15: Use of the x-refersTo extension example.

```
1 Pet:
2   x-refersTo: https://example.com/ontology/Pet
3   type: object
4   properties:
5     id:
6       x-refersTo: https://example.com/ontology/Id
7       type: integer
8       format: int64
```

Listing 6.16 contains the individuals resulting from the translation of the example of listing 6.15. A Pet_NodeShape is created for the Pet Schema Object (lines 1-4) and a Pet_id_PropertyShape is created for the id property of the Pet Schema Object (lines 6-10). In line 3 the Pet_NodeShape gets associated with the Pet Class specified in the x-refersTo value. Furthermore, in line 9 the Pet_id_PropertyShape is associated with the Id Class specified in the x-refersTo vlaue.

Listing 6.16: Individuals resulting from the translation of the Schema Object of Listing 6.15

```
1  ex:Pet_NodeShape a sh:NodeShape;
2    rdfs:label "PetNodeShape";
3    sh:targetClass <https://example.com/ontology/Pet>;
4    sh:property ex:Pet_id_PropertyShape .
5
6  ex:Pet_id_PropertyShape a sh:PropertyShape;
7    rdfs:label "Pet_id_PropertyShape";
8    openapi2:name "id";
9    sh:path <https://example.com/ontology/Id>;
10   sh:datatype xsd:long .
```

There is also a possibility that a Schema Object does not have a semantic value. An OpenAPI specification file could contain Schema Objects that are not widely used and are written for a specific purpose and service. Those Schema Objects most likely won't have a use for other developers. In this case, the x-refersTo extension can be used with a "none" value. If a "none" value exists, then the algorithm will not produce a class for that specific Schema Object or property. As a result, the *sh:targetClass* or *sh:path* properties will be missing from the NodeShape or PropertyShape respectively.

Listing 6.17: Use of the x-refersTo extension with a "none" value example.

```
Pet:
  x-refersTo: none
  type: object
  properties:
    id:
      x-refersTo: none
      type: integer
      format: int64
```

Listing 6.18 contains the individuals resulting from the translation of the example of Listing 6.17. Compared to the result of Listing 6.15 it can be seen that both *sh:targetClass* and *sh:path* properties are missing from the Pet_NodeShape and Pet_id_PropertyShape respectively, as a consequence of the "none" value of x-refersTo extension.

Listing 6.18: Individuals resulting from the translation of the Schema Object of Listing 6.17

```
ex:Pet_NodeShape a sh:NodeShape;
  rdfs:label "PetNodeShape";
  sh:property ex:Pet_id_PropertyShape .

ex:Pet_id_PropertyShape a sh:PropertyShape;
  rdfs:label "Pet_id_PropertyShape";
  openapi2:name "id";
  sh:datatype xsd:long .
```

6.5.2 Handling the x-kindOf semantic annotation property

Some OpenAPI data models (i.e. Schemas) have a narrower meaning than more generic ones, that they specialise. This is expressed using the x-kindOf semantic annotation property, which essentially denotes the current model as a subclass of the a more generic one. The example of Listing 6.19 contains a Dog Schema Object which contains the x-kindOf extension. This implies that the Dog model is a subclass of the Animal Class declared in the https://example.com/ontology domain

Listing 6.19: Use of the x-kindOf extension example.

```
1 Dog:
2     x-kindOf: https://example.com/ontology/Animal
3     properties:
4     packSize:
5       type: integer
6       format: int32
```

Listing 6.20 contains the individuals resulting from the translation of the example of Listing 6.19. In line 6, a Dog Class is created for the Dog_NodeShape. Then this class is associated with the Dog_NodeShape as seen in line 3 using the *sh:targetClass* property. Since the x-kindOf extension was added and has an Animal Class as value, the Dog Class created earlier will becomes a subclass of the Animal Class using the *rdfs:subClassOf* property as seen in line 7.

Listing 6.20: Individuals resulting from the translation of the Schema Object of Listing 6.19

```
1  ex:Dog_NodeShape a sh:NodeShape;
2    rdfs:label "DogNodeShape";
3    sh:targetClass  ex:Dog;
4    sh:property ex:Dog_packSize_PropertyShape .
5
6  ex:Dog a owl:Class;
7    rdfs:subClassOf <https://example.com/ontology/Animal> .
8
9  ex:Dog_packSize_PropertyShape a sh:PropertyShape;
10   rdfs:label "Dog_packSize_PropertyShape";
11   openapi2:name "packSize";
12   sh:path ex:Dog_packSize;
13   sh:datatype xsd:int .
14
15 ex:Dog_packSize a rdf:Property .
```

6.5.3 Handling the x-mapsTo semantic annotation property

The x-mapsTo property is used to state that a Schema Object element shares the same semantics with another Schema Object. The value of this property should be a reference to the semantically similar Schema Object. This property should not be confused with the $ref property. Listing 6.21 shows an example of a SecondPet Schema Object that contains the x-mapsTo extension that points to the Pet Schema Object. This declares that the two Schema Objects are similar semantically.

Listing 6.21: Use of the x-mapsTo extension example.

```
1  Pet:
2      type: object
3      properties:
4      id:
5        type: integer
6        format: int64
7  SecondPet:
8      x-mapsTo: '#/components/schemas/Pet'
9      type: object
10     properties:
11       secondId:
12         x-mapsTo: '#/components/schemas/Pet.id'
13         type: integer
14         format: int64
```

Listing 6.22 contains the individuals resulting from the translation of the example of Listing 6.21. Lines 1-6 contain the Pet_NodeShape for the Pet Schema Object. In line 6 the Pet Class is created for the Pet_NodeShape. Lines 16-19 contain the SecondPet_NodeShape for the SecondPet Schema Object. In line 18 the *sh:targetClass* property's object is the Pet Class because the SecondPet Schema Object targets the Pet Schema Object using the x-mapsTo property (the two Schema Objects are semantically equivalent). Using the same logic the SecondPet_secondId_PropertyShape's *sh:path* property has the Pet_id property as object.

Listing 6.22: Individuals resulting from the translation of the Schema Object of Listing 6.21

```
ex:Pet_NodeShape a sh:NodeShape;
  rdfs:label "PetNodeShape";
  sh:targetClass ex:Pet;
  sh:property ex:Pet_id_PropertyShape .

ex:Pet a owl:Class .

ex:Pet_id_PropertyShape a sh:PropertyShape;
  rdfs:label "Pet_id_PropertyShape";
  openapi2:name "id";
  sh:path ex:Pet_id;
  sh:datatype xsd:long .

ex:Pet_id a rdf:Property .

ex:SecondPet_NodeShape a sh:NodeShape;
  rdfs:label "SecondPetNodeShape";
  sh:targetClass ex:Pet;
  sh:property ex:SecondPet_secondId_PropertyShape .

ex:SecondPet_secondId_PropertyShape a sh:PropertyShape;
  rdfs:label "SecondPet_secondId_PropertyShape";
  openapi2:name "secondId";
  sh:path ex:Pet_id;
  sh:datatype xsd:long .
```

6.5.4 Handling the x-collectionOn semantic annotation property

The x-collectionOn semantic annotation property is used to indicate that a Schema Object is a collectio nof resources. A collection in OpenAPI is described using the array type, however it is common a collection's definition to be encapsulated within an object type with additional properties. The x-collectionOn property is used to denote the data types of the objects of the collection. Listing 6.23 defines a collection of Pet objects.

Listing 6.23: Use of the x-collectionOn extension example.

```
1  Pet:
2    type: object
3    properties:
4      id:
5        type: integer
6        format: int64
7  PetCollection:
8    x-collectionOn: pets
9    type: object
10   properties:
11     pets:
12       type: array
13       items:
14         $ref: '#/components/schemas/Pet'
```

Listing 6.24 contains the resulting from the translation of the Schema Objects of the example of Listing 6.23. The PetCollection Class becomes a subclass of *openapi2:Collection* Class (lines 21-22). The PetCollection_pets_PropertyShape that corresponds to the property "pets" of PetCollection, is defined as a member of a colleciton because its *sh:path* property has the *openapi2:member* value. Also the shape's *sh:node* property has the value of Pet_NodeShape which specifies the Node Shape that a value node conforms to.

Listing 6.24: Individuals resulting from the translation of the Schema Objects of Listing 6.23

```
ex:Pet_NodeShape a sh:NodeShape;
  rdfs:label "PetNodeShape";
  sh:targetClass ex:Pet;
  sh:property ex:Pet_id_PropertyShape .

ex:Pet a owl:Class .

ex:Pet_id_PropertyShape a sh:PropertyShape;
  rdfs:label "Pet_id_PropertyShape";
  openapi2:name "id";
  sh:path ex:Pet_id;
  sh:datatype xsd:long .

ex:Pet_id a rdf:Property .

ex:PetCollection_NodeShape a sh:NodeShape;
  rdfs:label "PetCollectionNodeShape";
  sh:targetClass ex:PetCollection;
  sh:property ex:PetCollection_pets_PropertyShape .

ex:PetCollection a owl:Class;
  rdfs:subClassOf openapi2:Collection .

ex:PetCollection_pets_PropertyShape a sh:PropertyShape;
  rdfs:label "PetCollection_pets_PropertyShape";
  openapi2:name "pets";
  sh:node ex:Pet_NodeShape;
  sh:path openapi2:member .
```

6.6 *Schema Object translation algorithm*

The recursive algorithm presented in Algorithm 13 is used to translate a Schema Object into ontology individuals. The translation process begins by checking the datatype of the schema object. A Schema Object may or not have a datatype. If a Schema Object does not have a datatype, that means it either has a reference to a reusable Schema Object in the OpenAPI components section, or it contains a keyword (allOf, oneOf, anyOf, not). If the Schema Object has an object datatype then we initialize a SHACL NodeShape, otherwise if the datatype is primitive (string, integer etc.) we initialize a SHACL PropertyShape. The use of SHACL NodeShape makes it possible to define constraints on node classes and validate nodes of these classes against the respective shapes. For example, a Person NodeShape for a Person Schema Object helps to ensure that person data sent to an API adhere to certain requirements. The use of SHACL PropertyShape allows to define constraints on individual RDF properties. In our case, they are used primarly as part of NodeShape to validate specific properties of nodes. For example, an age property of type integer of a Person Schema Object can be represented as an Age PropertyShape that is part of a Person NodeShape, which then will be used to validate the values given to that age property.

The second part of the algorithm contains the createNodeShape function, which is used to create a NodeShape for a Schema Object with object data type. This function's code shows the order we handle Schema Object extensions (x-kindOf, x-allOf, etc.), and Schema Object keywords (allOf, oneOf, etc.). After checking the extensions and keywords, the algorithm scans for Schema Object properties and calls the createPropertyShape function to handle the creation of PropertyShape(s) for each property. In the last line of the function we cache the NodeShape created for this particular Schema Object for later use (such as references to this Schema Object).

The third part of the algorithm contains the createPropertyShape function, which is used to create PropertyShapes for primitive or array data type Schema Objects. Like in createNodeShape function we first scan for keywords and then we check if the property has an array data type (requires different handling than primitive data types). Then we check for extensions and handle them using the handlePropertyExtensions function. Like in createNodeShape, we cache the produced PropertyShape for later use.

Algorithm 13 Schema Converter

```
1: procedure processSchema(Schema schema, String schemaName)
2:     dataType = schema → type
3:     if dataType = null then
4:         if hasKeywords(schema) then
5:             if hasOwner(schema) then
6:                 createPropertyShape(ownerName, schema, schemaName)
7:             else
8:                 createNodeShape(schema, schemaName)
9:         else if schema → $ref not null then
10:            cacheShape = cache.getShapeIndividual($ref)
11:            if cacheShape not null then
12:                if cacheShape is NodeShape then
13:                    shapeName = schemaName + "_NodeShape"
14:                    shapeName --rdf:type--> sh:NodeShape
15:                    shapeName --sh:node--> cacheShape
16:                else
17:                    shapeName = schemaName + "_PropertyShape"
18:                    shapeName --rdf:type--> sh:PropertyShape
19:                    shapeName --sh:node--> cacheShape
20:            else
21:                refSchema = fetchSchema($ref)
22:                refShape = translate(refSchema)
23:                if refShape is NodeShape then
24:                    shapeName = schemaName + "_NodeShape"
25:                    shapeName --rdf:type--> sh:NodeShape
26:                    shapeName --sh:node--> refShape
27:                else
28:                    shapeName = schemaName + "_PropertyShape"
29:                    shapeName --rdf:type--> sh:PropertyShape
30:                    shapeName --sh:node--> refShape
31:    else
32:        if dataType = "object" then
33:            createNodeShape(schema, schemaName)
34:        else if dataType = "string" then
35:            createPropertyShape(schema, schemaName)
36:        else if dataType = "number" then
37:            createPropertyShape(schema, schemaName)
38:        else if dataType = "integer" then
39:            createPropertyShape(schema, schemaName)
40:        else if dataType = "array" then
41:            createPropertyShape(schema, schemaName)
42:        else if dataType = "boolean" then
43:            createPropertyShape(schema, schemaName)
```

Algorithm 13 Schema Converter - Part2

```
44: procedure CREATENODESHAPE(Schema schema, String schemaName)
45:     shapeName = schemaName + "_NodeShape"
46:     shapeClassName = schemaName
47:     shapeName --rdf:type--> sh:NodeShape
48:     shapeName --rdfs:label--> schemaName+"NodeShape"

49:     schemaExtensions = schema → extensions
50:     if schemaExtensions not null then
51:         handleObjectExtensions(shapeName, shapeClassName, schemaExtensions)
52:     else
53:         shapeClassName --rdf:type--> owl:Class
54:         shapeName --sh:targetClass--> shapeClassName

55:     if hasKeywords(schema) then
56:         handleObjectKeywords(schema, shapeName, shapeClassName, schemaName)
57:     else
58:         for propertySchema in schema → properties do
59:             if propertySchema → $ref not null then
60:                 cacheShape = cache.getShapeIndividual($ref)
61:                 propertyShape = schemaName + "_" + propertyName + "_PropertyShape"
62:                 propertyExtensions = propertySchema → extensions
63:                 if propertyExtensions not null then
64:                     handlePropertyExtensions(propertyShape, propertyExtensions)
65:                 if cacheShape not null then
66:                     targetShape = cacheShape
67:                 else
68:                     refShema = fetchSchema($ref)
69:                     refShape = processSchema(refSchema, refSchemaName)
70:                     targetShape = refShape

71:                 propertyShape --rdf:type--> sh:PropertyShape
72:                 propertyShape --sh:node--> targetShape
73:             else
74:                 propertyShape = processSchema(propertySchema, propertyName, schemaName)
75:                 shapeName --sh:property--> propertyShape
76:     cache.addShapeIndividual("#/components/schemas/"+ schemaName, shapeName)
```

Algorithm 13 Schema Converter - Part3

```
77: procedure createPropertyShape(String ownerSchemaName, Schema schema, String schemaName)
78:     if ownerSchemaName is null then
79:         shapeIndividual = schemaName+ "_"+PropertyShape
80:         shapeClassName = schemaName
81:     else
82:         shapeIndividual = ownerSchemaName+ "_"+schemaName +"_"+PropertyShape
83:         shapeName = ownerSchemaName+ "_"+schemaName

84:     shapeIndividual --rdf:type--> sh:PropertyShape
85:     shapeIndividual --openapi2:name--> schemaName

86:     if hasKeywords(schema) then
87:         handlePropertyKeywords(schema, shapeIndividual, shapeClassName, ownerSchemaName)
88:     else
89:         if schema → type = "array" then
90:             Schema arrayItems = schema → items
91:             if arrayItems not null then
92:                 ref = arrayItems → $ref
93:                 if ref not null then
94:                     cacheShape = cache.getShapeIndividual(ref)
95:                     shapeIndividual --sh:Node--> cacheShape
96:                     shapeIndividual --sh:Path--> openapi2:member
97:                 else
98:                     inlineShape = processSchema(arrayItems, schemaName + "_Items", ownerName)
99:                     shapeIndividual --sh:Node--> inlineShape
100:                    shapeIndividual --sh:Path--> openapi2:member
101:        else
102:            schemaExtensions = schema → extensions
103:            if hasExtensions then
104:                handlePropertyExtensions(shapeIndividual, shapeClassName, schemaExtensions)
105:            else
106:                shapeClassName --rdf:type--> rdf:Property
107:                shapeIndividual --sh:Path--> shapeClassName
108:            set_XSD_Datatype(schema, shapeIndvidual)
109:        if ownerName not null then
110:            cache.addShapeIndividual("#/components/schemas/"+ownerName+"."+schem
    shapeIndividual)
111:        else
112:            cache.addShapeIndividual("#/components/schemas/"+schemaName, shapeIndividual)
```

Algorithm 13 Schema Converter - Part4

113: **procedure** HANDLEOBJECTEXTENSIONS(String shapeName, IRI shapeClassName, Map(String,Object) extensions)

114: finalClassName = null
115: **for** extensionName in extensions **do**
116: xReferesToFlag = false
117: extensionIRI = null

118: **if** extensionName == "x-kindOf" AND extensions.get(extensionName) == "none" **then**
119: xReferesToFlag = true
120: finalClassName = null

121: **if** extensionName == "x-kindOf" OR extensionName == "x-refersTo" AND !xReferesToFlag **then**
122: extensionIRI = IRI(extensions.get(extensionName))

123: **if** extensionName == "x-refersTo" AND !xReferesToFlag **then**
124: shapeName $\xrightarrow{sh:targetClass}$ extensionIRI
125: finalClassName = extensionIRI
126: **else if** extensionName == "x-kindOf" **then**
127: shapeClassName $\xrightarrow{rdf:type}$ owl:Class
128: shapeClassName $\xrightarrow{rdfs:subclassOf}$ extensionIRI
129: shapeName $\xrightarrow{sh:targetClass}$ shapeClassName
130: finalClassName = shapeClassName
131: **else if** extensionName == "x-mapsTo" **then**
132: reference = extensions.get(extensionName)
133: cacheShape = cache.getRefShape(referece)
134: **if** cacheShape not null **then**
135: cacheShapeClass = cache.getShapeClass(cacheShape)
136: **if** cacheShapeClass not null **then**
137: shapeName $\xrightarrow{sh:targetClass}$ cacheShapeClass
138: finalClassName = cacheShapeClass
139: **else**
140: refSchema = fetchSchema(reference)
141: refSchemaShape = processSchema(refSchema, refSchemaName)
142: refShapeClass(refSchemaShape)
143: **if** refShapeClass not null **then**
144: shapeName $\xrightarrow{sh:targetClass}$ refShapeClass
145: finalClassName = refShapeClass
146: **else if** extensionName == "x-collectionON **then**
147: shapeClassName $\xrightarrow{rdf:type}$ owl:Class
148: shapeClassName $\xrightarrow{rdfs:subclassOf}$ openapi2:Collection
149: shapeName $\xrightarrow{sh:targetClass}$ shapeClassName
150: finalClassName = shapeClassName

151: cache.addShapeClass(shapeName, finalClassName)

Algorithm 13 Schema Converter - Part5

```
152: procedure HANDLEOBJECTKEYWORDS(Schema schema, IRI shapeName, IRI shapeClass, String schemaName)
153:     allOf = schema → allOf
154:     anyOf = schema → anyOf
155:     oneOf = schema → oneOf

156:     if allOf not null then
157:         allOfList = createBlankNode()
158:         for Schema s in allOf do
159:             reference = s → $ref
160:             if reference not null then
161:                 cacheShape = cache.getShapeIndividual(reference)
162:                 if cacheShape not null then
163:                     allOfList.add(cacheShape)
164:                     cacheShapeClass = cache.getShapeClass(cacheShape)
165:                     if cacheShapeClass not null then
166:                         shapeClass = cache.getShapeClass(shapeName)
167:                         shapeClass --rdfs:subClassOf--> cacheShapeClass
168:                 else
169:                     refSchema = fetchSchema(reference)
170:                     refShape = processSchema(refSchema, refSchemaName)
171:                     allOfList.add(refShape)
172:                     refShapeClass = cache.getShapeClass(refShape)
173:                     if refShapeClass not null then
174:                         shapeClass = cache.getShapeClass(shapeName)
175:                         shapeClass --rdfs:subClassOf--> refShapeClass
176:             else
177:                 inlineSchemaProperties = s → properties
178:                 for propertyName in inlineSchemaProperties do
179:                     inlineShape = processSchema(inlineSchemaProperties.get(propertyName), schemaName + "_AllOfInline"+i)
180:                     allOfList.add(inlineShape)

181:         shapeName --sh:and--> allOfList
```

Algorithm 13 Schema Converter - Part6

```
182:    else if anyOf not null then
183:        anyOfList = createBlankNode()
184:        for Schema s in anyOf do
185:            reference = s → $ref
186:            if reference not null then
187:                cacheShape = cache.getShapeIndividual(reference)
188:                if cacheShape not null then
189:                    anyOfList.add(cacheShape)
190:                else
191:                    refSchema = fetchSchema(reference)
192:                    refShape = processSchema(refSchema, refSchemaName)
193:                    anyOfList.add(refShape)
194:            else
195:                inlineShape = processSchema(s, schemaName + "AnyOfInline"+i)
196:                anyOfList.add(inlineShape)

197:        shapeName --sh:or--> anyOfList

198:    else if oneOf not null then
199:        oneOfList = createBlankNode()
200:        for Schema s in oneOfList do
201:            reference = s → $ref
202:            if reference not null then
203:                cacheShape = cache.getShapeIndividual(reference)
204:                if cacheShape not null then
205:                    oneOfList.add(cacheShape)
206:                else
207:                    refSchema = fetchSchema(reference)
208:                    refShape = processSchema(refSchema, refSchemaName)
209:                    oneOfList.add(refShape)
210:            else
211:                inlineShape = processSchema(s, schemaName + "OneOfInline"+i)
212:                oneOfList.add(inlineShape)

213:        shapeName --sh:xone--> oneOfList
```

Chapter 7

Transforming Links into to OpenAPI ontology individuals

Links are a feature of OpenAPI version 3.0. Links are used to show the API user how a value returned by an operation can be used as input for other API operations. The use of this object is very rare in description files. Because of this reason we decided not to create a translation algorithm for this object. Instead we will show how such objects can be represented in the ontology.

In the example of Listing 7.1 the description file contains a "\users" POST operation which responds with an "id" value. In the "links" area of this operation the description declares that the "id" value returned as response can be used as a parameter in the operation with id "getUser" which is the "\users\userId" GET operation below.

The way we would represent a link inside an ontology is shown in Listing 7.2. We create the Response NodeShape (lines 1-3) for the schema of POST "\users" response along with a Property Shape for the "id" property it contains (lines 5-8). Next up we create a Property Shape for the "userId" parameter of GET "\users\userId" operation (lines 13-16). Lastly in order to show that the value of "id" can be used as a value of "userId", we connect the "userId" property individual and "id" property individual using the "owl:equivalentProperty" property (line 21) which declares that these properties receive the same values.

Listing 7.1: OpenAPI Link object example.

```yaml
paths:
  /users:
    post:
      summary: Creates a user and returns the user ID
      operationId: createUser
      requestBody:
        required: true
        description: A JSON object that contains the user name and age.
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
      responses:
        '201':
          description: Created
          content:
            application/json:
              schema:
                type: object
                properties:
                  id:
                    type: integer
                    format: int64
                    description: ID of the created user.
          links:
            GetUserByUserId:
              operationId: getUser
              parameters:
                userId: '$response.body#/id'
  /users/{userId}:
    get:
      summary: Gets a user by ID
      operationId: getUser
      parameters:
        - in: path
          name: userId
          required: true
          schema:
            type: integer
            format: int64
      responses:
        '200':
          description: A User object
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
```

LISTING 7.2: Theoritical result from the translation of the operation objects of Listing 7.1

```
ex:ResponseShape
      a sh:NodeShape ;
      sh:property ex:idPropShape  .

ex:idPropShape
     a sh:PropertyShape ;
     sh:path ex:responseId ;
     sh:datatype xsd:long .

ex:responseId
      a rdf:Property .

ex:useIdPropShape
     a sh:PropertyShape ;
     sh:path ex:userId ;
     sh:datatype xsd:long .

ex:userId
      a rdf:Property .

ex:userId owl:equivalentProperty  ex:responseId
```

Chapter 8

Ontology Evaluation

This section discusses the evaluation process. In Section 8.1, we analyze the translation of 10,000 specification files, including the success rate and the most common reasons for translation failures. In Section 8.2, we examine the complexity of converting these specification files and identify the main factors that affect the translation time and the size of the produced ontology. Section 8.3 outlines the benefits of using an ontology, highlighting the semantic difficulties that were easier to solve using an ontology. Finally in Section 8.4 we execute a set of SPARQL queries on a GraphDB triple-store database which contains the 9,212 ontology files to evaluate the ontology's efficiency and measure response times.

8.1 Validity Check

In the evaluation process we tested 9,954 Specification files downloaded from SwaggerHub. In order to test backwards compatibility we included files from OpenAPI version 2 up to version 3.1. The success rate of the translation of those files is 92,54% since we received an output of 9,212 ontology files that were then successfully imported into our Triple Store database.

The translation of 742 files was unsuccessful. For this reason we made sure to contain code that detects errors that may result in failure. Below we will analyze some of the most common mistakes that were detected during the evaluation process.

When writing an OpenAPI specification file, authors sometimes make one or more mistakes that can result to issues in API functionality and documentation. A prevalent issue is the inclusion of non-permitted characters inside schema names. Schema names should only contain alphanumeric characters and avoid special characters to ensure compatibility and avoid parsing and translation errors. For instance using "user info" as a schema name can lead to a translation error. Instead the author shoud opt for "userInfo" to be consistent.

Another common mistake is incorrectly referencing objects across different contexts. OpenAPI allows for object references to streamline definitions

and avoid redundancy, but these references must be contextually appropriate. For example, referencing a schema object from a parameter object is not permissible. Proper referencing ensures the API remains intuitive and its components correctly interrelated.

Circular references within schema objects will make translation algorithm enter an infinite loop. Circular dependencies occur when two or more schema objects reference each other in a loop, such as schema A referencing schema B, which in turn references schema A. To prevent this, avoid direct recursive references or restructure your schemas to eliminate the circular paths. Utilizing composition (e.g., allOf, oneOf, anyOf) judiciously can help manage complex schema relationships without falling into circular traps. Our error detection code does not handle this case.

In addition to these specific issues, it's crucial to regularly validate the OpenAPI document against the OpenAPI standard using tools like Swagger Validator. Validation tools can catch a wide range of errors, from simple typos to more complex logical inconsistencies, ensuring that the API specification adheres to best practices and standards. By meticulously crafting and validating the OpenAPI specification, authors can create robust, clear, and efficient API documentation that facilitates smooth development and integration.

8.2 Translation Complexity & Output Size

The time required for a translation process does not directly correlate with the size of the input file (whether it is minified or not). It is worth to note that larger files can sometimes be converted quicker than smaller ones. In our observations from 9212 translations, it is evident that files which take longer typically contain a higher number of Schema Objects.

As illustrated in Algorithm 13, converting Schema Objects is the most complex translation process among all OpenAPI Object translations. This complexity comes from the data types that Schema Objects can have, the different extensions and keywords they might include and the possibility of containing additional nested Schema Objects. These factors significantly increase the difficulty and time needed for Schema Object translation.

An example of a complex OpenAPI specification file can be found in this Swaggerhub Link. This OpenAPI specification file contains 48 predefined Schema Objects in components section and more Schema Objects defined inline within other objects such as Parameter and RequestBody Objects.

The output file size cannot be predicted because of various factors, such as the cache used in this work. The caching mechanism is designed to save produced individuals. For example a predefined RequestBody object will be converted only once and saved in cache. The produced individual will be reused when a reference to this RequestBody object is declared in one or more Operation Objects. By doing this we avoid creating multiple

individuals for the same object. A specification file that does not include references to identical objects will have a larger output file than a file that reuses objects with references. Generally it is observed that larger input files result in larger output file sizes.

Table 8.1 contains statistics from the translation of 9212 OpenAPI specification files.

Statistics			
	Input File Size (Bytes)	Output File Size (Bytes)	translation Time (ms)
Average	14163,64	43599,21	8,49
Maximum	1300439	4127129	1309

Table 8.1: Average & Maximum values for Input, Output File Size and Translation Time

It is worth to note that the largest output file in size was produced from the largest input file in size, but the file with the longest conversion time was not the biggest file in size.

8.3 Semantics

There are properties and relationships between data that can't be easily represented without an ontology. This section is going to provide some of the benefits of using semantic web technologies.

8.3.1 Inheritance and Polymorphism

In Section 6.4.3 we analyzed the handling of "allOf" keyword in Schema Objects. There are cases where model schemas share common properties. Instead of describing these properties for each schema repeatedly, a schema can be described as a composition of the common property set and schema-specific properties. This can be achieved using the "allOf" keyword. For instance, a "User" Schema Object could define a common property set, while a "SysAdmin" Schema Object could include additional schema-specific proprieties unique to system admins. By creating an ontology, we can represent the "User-SysAdmin" relationship by designating the "SysAdmin" Class as a subclass of the "User" Class similarly to the example of Listing 6.11. This allows for a more efficient representation of Schema Object relationships, which would not be as easily achievable without an ontology.

Also when a Schema Object is a combination of other Schema Objects that often results in multiple possible data types for that object. With the use of SHACL's logical properties (and, or, not, xone) we can represent a Schema Object as a NodeShape with multiple possible data types and validate data against that shape.

8.3.2 Reasoning

Reasoning tools can perform several tasks that improve the understanding and utilization of the knowledge within the ontology. A reasoner can perform a consistency check and verify that the relationships and definitions withing the ontology are coherent. Also based on the existing relationships and definitions a reasoning tool can infer new relationships between the data that were not explicitly stated before.

8.4 Response Time

To evaluate the capabilities of our ontology, we uploaded the ontology files generated from the translation of 10.000 OpenAPI Specification files into a GraphDB triple-store database. We used the same 10.000 OpenAPI Specification files to compare the results to [1] which stores service data in a MongoDB database. We then proceeded to execute SPARQL queries against the GraphDB triple-store and the equivalent OAQL2 queries against the MongoDB database. Below we list and discuss the results of some of the queries we executed.

Listing 8.1: SPARQL Query Example 1

```
PREFIX openapi2: <http://www.intelligence.tuc.gr/ns/v2/
    open-api#>
SELECT ?serviceTitle ?serviceVersion ?serviceDescription
WHERE{
    ?service a openapi2:Document.
    ?service openapi2:version ?serviceVersion.
    ?service openapi2:info ?serviceInfo.
    ?serviceInfo openapi2:title ?serviceTitle.
    ?serviceInfo openapi2:description ?serviceDescription
        .
}
```

Query of Listing 8.1 fetches the service title, version and description of every service ontology in the database.

Listing 8.2: Listing 8.1 OAQL2 Equivalent Query

```
SELECT s.title, s.version, s.description
FROM Service s
```

Response Time for Example Query 1		
	GraphDB (SPARQL)	MongoDB (OAQL2)
Average(ms)	111	178,2
Maximum(ms)	131	227
Minimum(ms)	101	154

Table 8.2: Average, Maximum and Minimum Response Time values between SPARQL and OAQL2

Results of Table 8.2 are similar for SPARQL and OAQL2.

Listing 8.3: SPARQL Query Example 2

```
PREFIX openapi2: <http://www.intelligence.tuc.gr/ns/v2/
    open-api#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT ?serviceTitle ?operationMethod ?pathName
WHERE {
    ?service openapi2:info ?serviceInfo.
    ?serviceInfo openapi2:title ?serviceTitle.
    ?service openapi2:supportedOperation ?
        serviceOperation.
    ?serviceOperation openapi2:method ?operationMethod.
    ?serviceOperation openapi2:onPath ?operationPath.
    ?operationPath openapi2:pathName ?pathName.
    FILTER (STR(?serviceTitle) = "API document medlink
        version 2")
}
```

Query of Listing 8.3 results contain every available path and its supported methods of a specific Service, in our case the Service title is "API document medlink version 2".

Listing 8.4: Listing 8.3 OAQL2 Equivalent Query

```
SELECT s.title, r.path, r.method
FROM Service s
JOIN Request r ON s
WHERE s.title = "API document medlink version 2"
```

Response Time for Example Query 2		
	GraphDB (SPARQL)	MongoDB (OAQL2)
Average(ms)	22	17,92
Maximum(ms)	22,6	20,8
Minimum(ms)	21,2	17,1

Table 8.3: Average, Maximum and Minimum Response Time values between SPARQL and OAQL2

Results of Table 8.3 are similar for SPARQL and OAQL2.

Listing 8.5: SPARQL Query Example 3

```
PREFIX sh: <http://www.w3.org/ns/shacl#>
PREFIX openapi2: <http://www.intelligence.tuc.gr/ns/v2/
    open-api#>
SELECT ?title ?pathName ?method
WHERE{
    ?service openapi2:info ?info.
    ?info openapi2:title ?title.
    ?service openapi2:supportedOperation ?operation.
    ?operation openapi2:method ?method.
    ?operation openapi2:onPath ?path.
    ?path openapi2:pathName ?pathName.
    ?operation openapi2:response ?response.
    ?response openapi2:content ?mediaType.
    ?mediaType openapi2:schema ?schema.
    ?schema sh:targetClass <https://schema.org/Person>.
}
```

The query in Listing 8.5 will yield API paths and methods that return data of the "Person" type. For the query to produce results, the API's response schema must be annotated with the x-refersTo extension and have the value "Person".

Listing 8.6: Listing 8.5 OAQL2 Equivalent Query

```
SELECT s.title, r.path, r.method
FROM Service s
JOIN Request r ON s
JOIN Response res ON r
JOIN Schema sc ON res
WHERE sc.x-refersTo =
"https://schema.org/Person"
```

Response Time for Example Query 3		
	GraphDB (SPARQL)	MongoDB (OAQL2)
Average(ms)	5,74	341,2
Maximum(ms)	7,55	347
Minimum(ms)	4,75	336

Table 8.4: Average, Maximum and Minimum Response Time values between SPARQL and OAQL2

In the results of Table 8.4 it is visible that queries on semantically annotated Schema Objects are answered much faster using an ontology and SPARQL language than queries done against the MongoDB using OAQL2 language.

Chapter 9

Future Work

Future work involves developing an extension of the OASL query language [12] for this new version of the OpenAPI ontology. A query language will make it easier for clients to search for specific services.

Additionally, development of service synthesis tools is a crucial next step. These tools will enable developers and clients with limited technical skills to easily connect services using ontology service data and create applications.

Lastly a great addition to this work would be the creation and management of a service database where users can search for specific service data.

Bibliography

[1] I. Apostolakis, N. Mainas, and E. G.M. Petrakis. Simple Querying service for OpenAPI Descriptions with Semantic Extensions. *Information Systems*, 117:102241, 2023.

[2] D. Martin at al. OWL-S: Semantic Markup for Web Services, November 2004. W3C Member Submission.

[3] F. Bouraimis. Instantiating OpenAPI Descriptions to the REST Services Ontology. Technical report, Diploma Thesis, School of Electrical and Computer Engineering, Technical University of Crete (TUC), Chania, Crete, April 2021.

[4] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, June 2007. W3C Recommendation.

[5] J. de Bruijn at al. Web Service Modeling Ontology (WSMO), June 2005. W3C Member Submission.

[6] J. Farrel and H. Lausen. Semantic Annotations for WSDL and XML Schema, August 2007. W3C Recommendation.

[7] M. Hadley. Web Application Description Language, August 2009. W3C Recommendation.

[8] E. Hamza, C. Izquierdo, J. Luis, and C. Jordi. Example-Driven Web API Specification Discovery. In *Modelling Foundations and Applications (ECMFA 2017)*, pages 267–284, Marburg, Germany, July 2017.

[9] M. Klusch. Semantic web service description. In M. Schumacher, H. Schuldt, and H. Helin, editors, *CASCOM: Intelligent Service Coordination in the Semantic Web*, pages 31–57, Basel, Switzerland, July 2008.

[10] H. Knublauch and D. Kontokostas. Shapes Constraint Language (SHACL), July 2017.

[11] Holger Knublauch and Dimitris Kontokostas. Shapes constraint language (shacl). *W3C Candidate Recommendation*, 11(8):1, 2017.

[12] N. Lagogiannis, N. Mainas, C. Tsinaraki, and E. G. M. Petrakis. OASL: SPARQL Query Language for OpenAPI Ontologies. In *37th Intern. Conf. on Advanced Information Networking and Applications*

(AINA 2023), pages 303–317, Juiz de Fora, Brazil, 29-31 March 2023, 2023.

[13] M. Lanthaler and C. Gütl. A Vocabulary for Hypermedia-Driven Web APIs. In *Workshop on Linked Data on the Web (LDOW 2013)*, Rio de Janeiro, Brazil, May 2013.

[14] N. Mainas, F. Bouraimis, A. Karavisileiou, and E. G. M. Petrakis. Annotated OpenAPI Descriptions and Ontology for REST Services. *International Journal on Artificial Intelligence Tools*, 32(6):2350017, September 2023.

[15] N. Mainas and E. Petrakis. Soas 3.0: Semantically enriched openapi 3.0 descriptions and ontology for rest services. In *IEEE Intern. Conf. on Semantic Computing (ICSC 2020)*, pages 207–210, San Diego, California, 2020.

[16] N. Mitra and Y. Lafon. SOAP Version 1.2 Part 0: Primer (Second Edition), April 2007.

[17] Wardani Muhamad, Suhardi, and Yoanes Bandung. Transforming openapi specification 3.0 documents into rdf-based semantic web services. *Journal of Big Data*, 9(1):55, 2022.

[18] F. A. Musyaffa, L. Halilaj, R. Siebes, F. Orlandi, and S. Auer. Minimally Invasive Semantification of Light Weight Service Descriptions. In *IEEE International Conference on Web Services (ICWS 2016)*, pages 672–677, San Francisco, CA, USA, June 2016.

[19] S. Schwichtenberg, C. Gerth, and G. Engels. From Open API to Semantic Specifications and Code Adapters. In *IEEE International Conference on Web Services (ICWS 2017)*, pages 484–491, San Francisco, CA, USA, June 2017.

[20] A. Sheth, K. Gomadam, and J. Lathem. SA-REST: Semantically Interoperable and Easier-to-Use Services and Mashups. *IEEE Internet Computing*, 11(6):91–94, Nov.-Dec. 2007.

[21] Amit P Sheth, Karthik Gomadam, and Jon Lathem. Sa-rest: Semantically interoperable and easier-to-use services and mashups. *IEEE Internet Computing*, 11(6):91–94, 2007.

[22] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, P.A. Champin, and N. Lindström. Json-ld 1.1: A json-based serialization for linked data, July 2020.