Transformation from OntoUML models to the OpenAPI Specification

KOEN DE JONG, University of Twente, The Netherlands

This paper addresses a transformation for integrating OntoUML models, an ontology-driven conceptual modelling language, with the OpenAPI Specification for RESTful APIs. Despite OntoUML's expressiveness in capturing complex relationships within domains, a significant gap hinders effective collaboration between ontologists and developers. A systematic transformation process, preserving semantic richness, and providing user-friendly tools helps to bridge this gap. The result combines a literature review, practical implementation, and qualitative insights to achieve a transformation which can help with understanding and collaboration within software development.

Additional Key Words and Phrases: Ontology-based transformation, OntoUML, UFO, REST API, OpenAPI, JSON:API

ACM Reference Format:
Koen de Jong. 2023. Transformation from OntoUML models to the OpenAPI Specification. In . ACM, New York, NY, USA, 9 pages.

1 INTRODUCTION

Many organisations, whether they are developers at corporations or researchers at public institutions, use ontologies to represent an ever-evolving landscape of information systems. They use them to make the nature and structure of engineered systems more explicit [17] while helping us to comprehend the different complexities that the stakeholders and environment dictate for us [18]. Several ontology languages have been developed for this reason and one of them is OntoUML, an extension of the Unified Modeling Language (UML) and an ontology-driven conceptual modelling language designed to reflect ontological distinctions of the upper-level ontology called Unified Foundational Ontology (UFO). It serves as a framework for capturing intricate relationships within any given domain. Currently, OntoUML finds successful applications across various domains at present [9] and more tools are being developed to support its utilization and further enhance its impact.

The current landscape of information systems demands a mutually beneficial relationship between ontological models and the practicalities of software development. Despite the rich expressiveness that OntoUML provides in capturing intricate relationships between concepts, a considerable gap persists in effectively translating this ontological knowledge into the language of developers, specifically for this research, the OpenAPI Specification. A study in 2021 [5] analyzes the use of ontologies during the development of tools that are used for building APIs. We find that in many cases ontologies are not included in the design process, highlighting the gap which represents a barrier to achieving effective collaboration. Developers, tasked with creating functional software systems, often find themselves grappling with the challenge of integrating ontological models into their workflows. The OpenAPI Specification [2] stands as the basis of a communication language for web developers, offering a standardized and interpretable text for both developer and machine. Its widespread adoption has streamlined collaboration among developers and enhanced the accessibility of APIs [3]. However, creating an API of the model is often a lot of work and a small change in the model might have a large impact on the API [6]. This impedes collaboration because the developers often have little knowledge of ontology models while ontologists spend countless hours refining them [9].

The goal of this research is to develop a system for transforming OntoUML models into the OpenAPI Specification, thereby contributing to filling the gap. The aim is to ensure that the semantics of OntoUML are harnessed within the widely adopted OpenAPI Specification. The goal is reached by the achievement of the sub-goals:

- Design a process for transforming OntoUML models into the OpenAPI Specification. This includes defining clear guidelines and constraints to capture and ensure the preservation of the semantic richness.
- Illustrate the transformation process by a use case example.
- Evaluate the implemented transformation process.

In the upcoming sections of this paper, we will navigate through the background research which lays the foundation for the approach of the transformation and explains the tools that are used. After this, section 3 analyses which requirements a successful transformation should adhere to. The main contribution of this paper is given in section 4, which is divided into two parts: abstraction and mapping. When applying abstraction we reduce the original graph to not overcomplicate the result, then we convert the reduced graph to OpenAPI with distinct mappings. A use case illustration is given next to the entire transformation to supply you with visual context. Next, section 5 compares the results presented there with relevant related work and we conclude in section 6 with some final considerations and future work ideas.

2 RESEARCH BASELINE

2.1 OntoUML

The need for ontologies, as described by Mizoguchi and Borgo in [11], is indicated by the significance of understanding a system. They show the interaction and dynamic functionality between entities within the models as well as their relations. The Unified Foundational Ontology (UFO) [8] is one of these ontologies. It is a framework developed to provide dynamic entities and their interactions and relies on micro-theories that address a range of conceptual modelling topics. Together, they cover the taxonomy of objects, the nature of part-whole relationships and the classification of events and roles among subjects. [16] shows that the UFO is currently the second most used ontology, but is also being adopted by more and more researchers as the fastest-growing ontology. OntoUML is an ontology-driven conceptual modelling language that extends UML

TScIT40, Februari 02, 2024, Enschede, The Netherlands

class diagrams by defining a set of stereotypes that reflect UFO ontological distinctions into language constructs. Between these stereotypes, we define more intricate relations that give more meaning than many standard associations provided by UML. OntoUML has some important distinctions between entities:

2.1.1 Types and individuals. Within the context of OntoUML, a type is something that classifies a thing. They are abstract ideas that help us classify the things around us. For example, a person is a type, but you and I are not. We are instantiations of that type or so-called individuals. OntoUML does not define individuals, only types.

2.1.2 Identity. Identity focuses on how entities establish and maintain their distinctiveness within a system. This involves understanding the criteria and characteristics that define the identity of entities and how these criteria influence their representation and behaviour within the ontology. We talk about types that provide identity and those that do not. For example[1], when distinguishing between a complete statue or the same statue that misses an arm, most would argue that the statue's appearance has changed, but its identity has not. It is still the same statue, it just misses an arm. Identity classes are also referred to as sortals, and consequently, classes that do not provide identity are non-sortals.

2.1.3 Rigidity. When we look at rigidity in OntoUML, the emphasis is on capturing the degree to which an entity's properties and identity remain constant or change over time. Rigidity is crucial in understanding the dynamic nature of entities within a system and how their attributes may evolve or persist under various conditions. We distinguish between types that are rigid and types considered non-rigid. If a person grows older, is that person still the same person? Well yes, but what if we look at a student? A person can be a student forever, but that role usually changes over time.

2.2 Ontology abstraction

Before the transformation of a graph to any desired structure, it is good practice first to reduce it to a minimized version of that graph. That way, we do not overcomplicate our final results and ensure an easier understanding of the underlying patterns and relationships within the data. Abstraction plays a crucial role in this process, as it allows us to focus on the essential elements of the graph while discarding unnecessary details. Multiple approaches have been created for the abstraction of ontologies.

One particular approach [10] suggests four rules for different patterns within OntoUML. This paper builds on the ideas proposed by Egyed in [4]. His approach proposes some standardized replacements in a small subsection of a UML graph. Abstraction of the graph can be substantial if it is possible to replace enough patterns while keeping the graph as close to the expansion as possible. In general, every rule abstracts a small structure into a smaller form. They are formed according to two principles:

- *Objects* are prioritized over *Relators* because the latter inherently relies on the existence of the former.
- The most important object types are those that provide identity and are rigid because they establish a stable foundation for the essentials of the model and aspects of the domain.

Other approaches such as [15], or extending approaches as [14] also use abstraction for the minimization of graphs in OntoUML, but the suggested rules by [10] are used as the methodology because the research limits itself to the constructs abstracted there. Thus, the rules of Guizzardi et al. lay a foundation for us to build a transformation process on in section 4, and they are tackled shortly down here one by one:

2.2.1 Abstracting Relators. A *Relator* defines a more intricate relation and always relies on other individuals to exist. In the abstraction of a *Relator*, the *Relator* is removed and only the derived relations are left. Between them then exists not a *Material* relation, but a *Formal* relation with the same cardinalities. This is possible because the relation still exists and is functional for an API, but this does introduce an ambiguity problem as highlighted in [8] for conceptual models.

2.2.2 Abstracting non-sortals. Classes that can have multiple identity principles are called non-sortals. They capture properties that are shared by individuals of different kinds. This second rule replicates all attributes of the non-sortal to the specializing types while removing the non-sortal. Any relations of the non-sortal will also be replicated and receive an annotation of the name of the non-sortal. This achieves less complexity in the model but may add duplication of properties.

2.2.3 Abstracting sortals. Although sortals are the building blocks of an API and the most important classes within OntoUML it is possible to abstract them. Through the abstraction of *Subkinds*, *Phases* and *Roles* we try to concentrate information around the *Kinds* in our model. As opposed to lowering the attributes as seen above, they are lifted towards the general stereotype. Relations are also linked to the general class, but we could lose cardinality constraints defined in the original model by doing so.

2.2.4 Abstracting partitions. Phase and Subkind partitions are generalisation sets defined in OntoUML and are a popular design pattern for specifying the categories of a property of some class. The difference lies in their rigidity. Subkinds are static and their instances cannot change between the elements of the generalisation set. Compare this to dynamic phases which can be updated over time. The 4th rule states that we can abstract generalisation sets into enumerations by lifting the attributes of the classes and basing the literals of the enumeration on the names of the classes. The name of the enumeration should be retrieved from the name of the generalisation set or composed from the different literals.

2.3 OpenAPI

The goal of OpenAPI is to enhance API development by providing a standardized way an Application Programming Interface (API) communicates. OpenAPI defines a schema for one or multiple APIs. It describes certain information about the API itself, such as a title, description and version, but more importantly, it describes the API's endpoints. Multiple HTTP methods can be defined for endpoints,

[1] This example is reused from https://ontouml.readthedocs.io/en/latest/theory/identity.html

which can all have possible responses to the request. Any endpoint can also have parameters either in the path, query, headers or body. The most straightforward way to standardize responses is to use schema objects that might have different properties and types. It is also possible to define combinations of types and have required properties.

OpenAPI also describes security requirements and mechanisms, which mostly cover how to handle servers and authorization. We do not take this into regard because including authentication and permissions within the transformation is not one of the key points within the transformation and is thus beyond the scope of this paper.

2.4 JSON:API

While OpenAPI provides a systematic framework for the expected communication the API provides, it can be integrated with JSON:API to make sure responses are also standardized. JSON:API is designed to minimize both the number of requests and the amount of data transmitted. It is also readable by humans and uses a schema to communicate.

The root structure defines either data or errors. We focus on the data transmitted because errors are part of the API and not of the model that describes the relations between objects. The data is a resource object or a list of resources. Every resource object has a type and identifier *id* and may have attributes and relationships. Relationships within the object only define the type and id. The actual resource it links to is transmitted in an included section on the root object. That way, if multiple objects link to the same relation, we do not duplicate data. JSON:API also defines links for every resource and its relationships. These links are useful for querying the exact data from that particular object.

3 REQUIREMENTS

To successfully transform any OntoUML model to the OpenAPI specification, requirements are set that we wish to achieve. These requirements can be perceived as the foundation and ensure a comprehensive and effective transformation process if they are met. To keep this research within its proper bounds, only the stereotypes *Kind, Quantity, Collective, Subkind, Role, Phase, Category, Mixin, RoleMixin, PhaseMixin and Abstract* are transformed.

To achieve completeness, the finalized (transformed) OpenAPI specification should be compatible with the OntoUML modelling standards. That entails maintaining the integrity and properties of the ontological constructs and relationships defined in the original OntoUML diagram [7] throughout the process. Loss of information or deviation from OntoUML standards should be avoided and minimized where possible to ensure a seamless and recognizable transition between the representations.

The process should also accurately try to capture all relevant constraints, relationship cardinalities and semantic nuances embedded within the model. This is to some extent, because when flattening the original model information can get lost and not everything can be converted to the OpenAPI specification.

The comprehension of any ontological model without knowing what it contains is crucial. What is meant by this, is that the transformation should be created even though it is unknown what models it will be used for. It should be flexible and accommodate variations in OntoUML models. There are quite a lot of recurring design patterns, but different modelling conventions also pop up often. The objective is to translate all diverse elements into the OpenAPI specification and ensure adaptability and consistency throughout the OntoUML practices without compromising the semantics of the result.

Additionally, the transformation process should support OntoUML models of varying sizes and complexities. Whether dealing with focused or intricate models, the process should efficiently generate the corresponding OpenAPI specification without significant shortcomings. Complexity management of models should also extend to the transformation process, not to limit its results, but to create a comprehensive and usable outcome.

Next to that, given the dynamic nature of OntoUML models, the transformation tool must be maintainable to accommodate changes and updates in the OntoUML or OpenAPI specification. This requires an extensible architecture that allows the integration of new constructs or revisions without a necessary overhaul of the transformation tool.

By addressing these requirements, the transformation tool should provide a robust and reliable mechanism for converting OntoUML models into OpenAPI specifications. This, in turn, facilitates a seamless and integrated transition between ontological modelling and API development processes.

4 APPROACH

To showcase how the transformation works, the approach will be demonstrated by a running example throughout the whole section. The complete diagram can be found in the Appendix under figure 6. It is an example based on a university structure within the OntoUML models GitHub repository[2].

4.1 Abstraction

The initial phase of the transformation process involves abstraction. We base the approach for abstraction on four rules of a paper written in 2019 by Giancarlo Guizzardi et al. [10]. The adaptations to the approach are made to make the transition to an API easier, most importantly the attributes and relation of the class are taken into account. After all, what is a class within an API if it does not hold actual data? For the abstraction visualisation the OntoUML plugin within Visual Paradigm[3] is used.

Hereafter we present a set of graph-rewriting rules based on [10] and where some classes may lose or change stereotypes. Stereotypes should during the abstraction process be looked at as a guideline which shows their identity and rigidity and can be used to eventually transform a model to OpenAPI. The abstraction result should not be viewed as a valid OntoUML.

Another difference is that abstraction can be affected by previous abstraction methods. Usually, generalisations and relationships can hinder a step from completion until later. The flow for abstraction

[2]https://github.com/OntoUML/ontouml-models/tree/master/models/university-ontology

[3]Visual Paradigm is a program that can be used to create many different modelling diagrams. https://www.visual-paradigm.com/

can be described as follows: Rules 1 to 4 from section 2.2 are repeatedly applied until the previous graph is the same as the graph before it. That is to say, the graph can no longer be abstracted any further.

Let us start by abstracting all *Relators* as proposed by 2.2.1. We make a distinction here between *Relators* that have attributes, generalisations or those that are ternary, meaning they have more than two relationships(1), and all others (2). If the *Relator* fits in the second category, it is removed as previously suggested. If on the other hand, the *Relator* conforms to the second case, then the *Relator's* stereotype is changed to a kind, meeting the same identity and rigidity principles and the *Material* is removed. Instead, the *Mediations* are transformed to *Formal* relationships and keep the same cardinalities. Figure 1 shows an example of a *Student*, *Enrollment* and *Course*. Because the *Relator Enrollment* contains attributes, it cannot be removed and is thus changed to a *Kind*. The *Material* relation is removed and the *Mediations* become *Formals*.

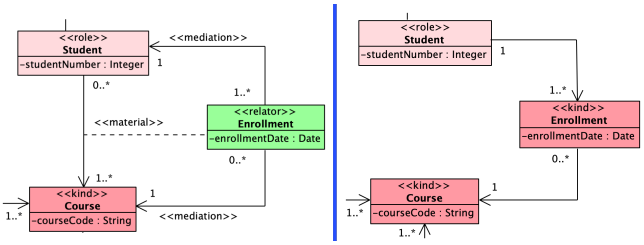

Fig. 1. Relator Abstraction

For the second, third and fourth rule, we stick to the solution from sections 2.2.2, 2.2.3 and 2.2.4. Additionally, every attribute that moves between classes is prefixed by the name of the original class, making the name clearer and easier to trace back to its original class. You also prevent duplicate cases by doing so. Let us look at a couple of structures that show the abstraction process.

Let us start with the removal of a non-sortal, in our case only the *Agent* class of stereotype *Category*. *Agent* here is a representational entity that should have a name. The abstraction shows how the attribute *name* is brought down to the generalisation specifiers *Person* and *Organisation*, prefixing it with the name of the *Agent*. The result is shown on the left-hand side of figure 2 in *Person*, which now contains an attribute *agentName*.

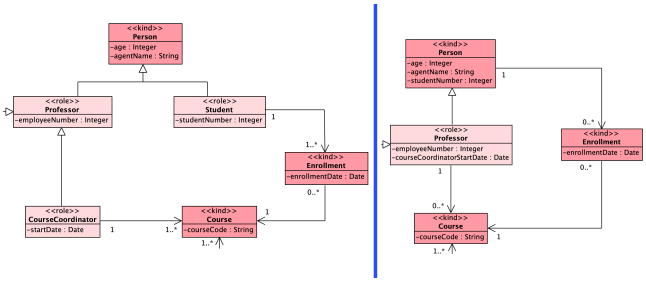

Fig. 2. First Sortal Abstraction

In a different step, but shown in the same figure, sortals are abstracted. There are three occurrences:

- *Professor*: *Professor* can not yet be abstracted because of its generalisation set *Status* and general role to the specific *CourseCoordinator*. First, *CourseCoordinator* is abstracted, then we look at *Professor* once more.
- *Student*: This *Role* can be abstracted into the *Kind* called *Person*. We lift the attributes of *Student* and connect the relationships it holds to *Enrollment* to *Person*. That does mean that not the cardinality changes. *Student* had a requirement to have at least one enrollment, but that semantic is sadly lost when it is flattened to *Person*.
- *CourseCoordinator*: The same principle as above can be applied to *CourseCoordinator*. First, the attributes are lifted, and then the relation is changed because not every *Professor* is a *CourseCoordinator*.

The changes that are made can be seen in figure 2. The distinction between classes removed in the abstraction process and the original class, as seen in the example, lies in the presence of certain attributes. If *Professor* has one or more courses, we know that they are a *CourseCoordinator*.

Following this, the two generalisation sets are flattened. We look at both cases separately:

- *Status*: This *Phase* partition is abstracted to an enumeration by taking the names of the *Phases* and using them as literals for the enum. The status of the professor can change over time because of the non-rigid qualities of *Phases*. *Status* has no other dependencies or relations, so the partial abstraction is complete.
- *Type*: The organisation type is abstracted from a *Subkind* partition. That means that it is a static enumeration and although this is not shown in the graph, within OpenAPI this is eventually transformed into a read-only attribute. The relation that existed between *University* and *Faculty*, is now a relation to itself, also with the cardinality changed because not every organisation (only *Universities*) has two *Faculties*, but only some do.

Both results can be seen in figure 3.

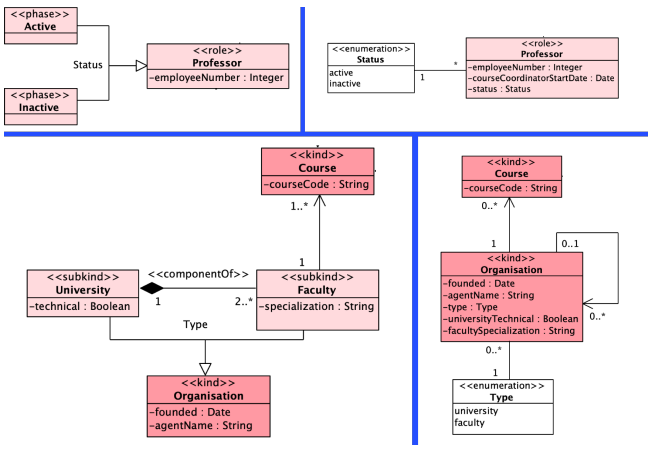

Fig. 3. Generalisation Sets Abstraction

All rules were abstracted, so we start over and see if any changes can be made. Because *Professor* has no generalisation to *CourseCoordinator* anymore, it is now possible to abstract this using a sortal abstraction mentioned in 2.2.3. *Professor* is flattened into *Student*, and noteworthy: the enumeration *Status* is not required anymore, because only professors have this attribute. Figure 4 shows this.

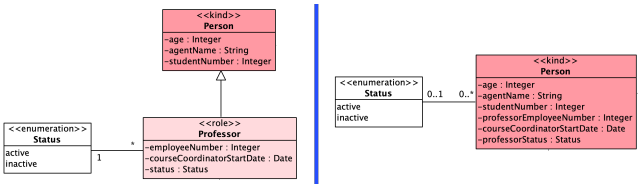

Fig. 4. Second Sortal Abstraction

There are no classes that can be abstracted, and thus the abstraction process is now complete. A finalized graph can be seen in the appendix as figure 7.

4.2 Mapping

After the abstraction, every OntoUML element left will be mapped to some part within OpenAPI. We distinguish between the different elements that are left within the model, namely classes, attributes and relationships. A proof of concept transformation is built within a fork of the ontouml-js repository on github[4]. In this repository, there are already some other transformations, for example from OntoUML to OWL. It also has tools that easily parse models and show the structure of class properties, showcasing its potential and the interest of others in this research area.

The basis of any OpenAPI schema is the following:

```yaml
openapi: '3.0.0'
info:
  title: UniversityModel
  version: 1.0.0
# objects beneath are filled by the transformation
components:
  schemas:
  responses:
paths:
```

Because the space for this paper is limited, the next examples will not show the entire example but individually refer to where they fit in the schema.

4.2.1 Classes. Classes after abstraction are transformed into schema objects that directly reflect the name of the Class. There are two exception cases and a general one:

- *Datatype Stereotype*: All Datatypes are evaluated during the attribute mapping in 4.2.2, and thus skipped here.
- *Enumeration Stereotype*: Enumerations are mapped to a string schema type with the property enum which contains the literals of the enum. Multiple values of the enumeration are impossible because it is a disjoint and complete class.
- *For all other cases*: they are transformed into an object schema, where properties can be added later. (4.2.2)

A partial example shown earlier is continued here:

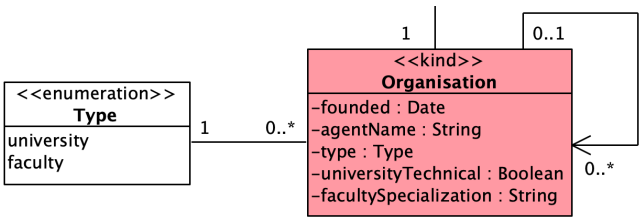

Fig. 5. Mapping Example

When the class transformation is applied, the result is the following:

```yaml
Organisation:
  type: object
Type:
  type: string
  readOnly: true # because of its static origin of Subkinds
  enum:
    - university
    - faculty
```

As you can see above, both classes keep their name and become schema objects. *Organisation* becomes an object schema where attributes can be added later because it is a *Kind* and thus falls in the other category, while *Type* is mapped to the enumeration schema.

4.2.2 Attributes. The information of our system is stored in attributes of schema objects. OpenAPI defines six types, where the first four are primitive:

- string (*primitive*)
- integer (*primitive*)
- number (*primitive*)
- boolean (*primitive*)
- object
- array

Every attribute is assigned to the schema it belongs to and the name of the datatype is compared with the types above and other schemas. If it does not match, the type is checked against the names of other schema objects. It might be an *Enumeration* for example. If nothing is found, we default to a string because it is the most dynamic and able to represent other types. If the name of the datatype contains an opening and closing square bracket, the type is assumed to be an array and the rest of the name is matched as described before. The string type can also fitted with a formatter to display dates and emails for example.

We know that some attributes are not required, usually because they were lifted from a specialized class to a general one. In the OpenAPI specification, this can be denoted with the keyword `nullable`. All other requirements are marked as required.

Attributes are not placed directly on the root object but are nested into other property attributes. This fits the JSON:API description of how attributes should be formatted in section 2.4.

[4]https://github.com/OntoUML/ontouml-js

The first three attributes of *Organisation* are transformed from figure 5 to show how an attribute is transformed.

```yaml
Type: ... # Type is defined previously
Organisation:
  type: object
  properties:
    attributes:
      type: object
      required:
        - founded
        - agentName
      properties:
        founded:
          type: string
          format: date
        agentName:
          type: string
        type:
          $ref: '#/components/schemas/Type'
          nullable: true
```

Here can be seen that *Type* has no type, but references the enumeration. This way, our result is understandable and users can see that any reference to *Type* should conform to the specified enum.

4.2.3 Relations. As mentioned before, relationships in OntoUML are pivotal in capturing associations between entities. While there are many different stereotypes for relationships, we maintain our focus on their existence and cardinalities. The reason for this is that we minimize relations to the attribute relationships on schemas and their stereotypes have no impact on the actual OpenAPI result. Some validations are only part of the API, not its representation and communication medium.

A distinction made for when a relation is a resource or when it should be an array of said resources is based on the cardinality of the relation. Within ontouml-js four cases pop up:

- zero-to-one: This emerges in the *Organisation* previously mentioned. It should have a parent *University* if the organisation is a faculty. *University* is in this case not required but can specify a relation to *University*.
- zero-to-many: This relation is often seen because the abstraction of generalisation leads to some properties not being required anymore. A *Person* can have multiple, or an array, of *enrollments*.
- one-to-one: The resource is required and a single resource. An *Enrollment* should for example always have exactly one *Person* and *Course*.
- one-to-many: It is required and an array, even if this array only contains one element. During abstraction, this is often replaced because specializations that define these relations get abstracted into the super-class.

Relations are defined within a property relationships, in line with how attributes are also defined to fit the JSON:API schema. An important note here is that a reference is made to the related object schema for clarity. In reality, any object would only be dispatched here with its id and type to be retrieved from the JSON:API included section.

Although not seen in the abstracted diagram, relations that perhaps specify a different relation with a minimum of required items, for example, that a car should have a minimum of 4 wheels. In this case, we can make use of the *minItems* and *maxItems* properties of the array (and object) type.

```yaml
Wheel: ...
Car:
  type: object
  properties:
    relationships:
      type: object
      required:
        - wheels
      properties:
        wheels:
          type: array
          minItems: 4
          items:
            $ref: '#/components/schemas/Wheel'
```

We continue with the example used before in figure 5 where the relation between *Organisation* and *Course* is transformed. Because an annotation has been made that only faculties can have courses, *Course* receives a relationship called faculty which references an *Organisation*. On the other end of the relation, we see that only when an organisation is a faculty it can have courses. The relation *Organisation* has to itself is left out of the example below.

```yaml
Course:
  type: object
  properties:
    relationships:
      type: object
      required:
        - faculty
      properties:
        faculty:
          $ref: '#/components/schemas/Organisation'
Organisation:
  type: object
  properties:
    ... # previously included properties
    relationships:
      type: object
      properties:
        facultyCourses:
          type: array
          nullable: true
          items:
            $ref: '#/components/schemas/Course'
```

Next should be noted that OntoUML can have ternary relations, or in terms of its properties: A relation with more than two endpoints. Although rarely seen[5], they cannot be skipped. A ternary relation should be transformed into an object schema with each endpoint of the relations as an attribute and the respective cardinality. The endpoints can then reference back to the newly created object schema.

4.2.4 Responses. Response standardization is done using JSON:API. For every object type in our schema, we compose a path to be retrieved in bulk and to create objects. Retrieving, updating and deleting objects can also be done individually. Using JSON:API we create the responses according to the specification, generating examples by recursively going through the schemas references and setting examples. JSON:API also standardizes a way to filter and create information, but that is left out to keep the examples simple to understand.

[5] An example of a ternary relation can be seen in [13]

The example beneath shows a fragment of the OpenAPI specification. It uses a response component for a GET request for multiple *Organisations*. The other requests are incomplete, but show the endpoints where information can be retrieved from. They are implemented in the same way as the collective GET.

```yaml
components:
  schemas:
    Base:
      type: object
      properties:
        id:
          type: integer
        type:
          type: string
    Organisation: ... # previously defined
  responses:
    Organisations:
      content:
        application/json:
          schema:
            type: object
            properties:
              data:
                type: array
                items:
                  allOf:
                    - $ref: '#/components/schemas/Base'
                    - $ref: '#/components/schemas/Organisation'
              included:
                type: array
                items:
                  type: object
              links:
                type: object
paths:
  '/organisations':
    get:
      responses:
        '200':
          $ref: '#/components/responses/Organisations'
    post: ...
  '/organisations/{id}':
    get: ...
    put: ...
    delete: ...
```

5 RELATED WORK

Several initiatives have aimed to encourage the integration of ontology languages into development practices. Three approaches map the Web Ontology Language (OWL) to the OpenAPI specification.

One of them is Ontology-Based APIs (OBA) [6], an ontology-based framework that generates complete server APIs with OpenAPI specifications for OWL knowledge graphs. Different from the implementation presented here, the mapping is direct and without abstraction. That means generalisations are not flattened and thus expressed by an allOf structure in OpenAPI. OBA defines an explicit mapping but is limited to the simple translation of classes, subclasses and their properties and types.

[12] improves this by extending the work of OBA (and implementation of it in OBA) and contributes a more complex mapping improving overall coverage by the use of non-primitive types, cardinalities and required values. Boolean operations such as intersections, unions and complements are also supported, but their combinations are not.

A large difference is that neither uses an underlying framework for the communication of data. A completely separate approach [1] uses JSON-LD, a framework with the same purpose as JSON:API, and also transforms OWL to OpenAPI, but misses an explicit mapping making it hard to determine its effectiveness. [12] concludes by manual inspection that the mapping is similar but seems to miss the boolean class operations mentioned earlier.

Although these related works and this paper both use an ontology-based model and result in an OpenAPI Specification, the approach differs quite radically. The abstraction process takes up a big part of this change, where we assume that the cost of lost semantics is gained in the clarity of the result. Though a transformation without the intervention of a human might abstract key features that simplicity cannot capture, an API structured with every original class becoming an endpoint is cluttered and leads to increasing API requests.

6 CONCLUSION

In conclusion, this paper presents an approach to transform OntoUML models into OpenAPI specifications for REST API development. The overarching goal is to establish a system for transforming OntoUML models into the OpenAPI Specification, thereby facilitating a seamless integration process. This integration aims to enhance collaboration between ontologists and developers, ensuring that the rich expressiveness of OntoUML is effectively harnessed within the widely adopted OpenAPI Specification.

Clear guidelines and constraints are defined to capture and preserve the semantic richness of OntoUML during the transformation process, which involves abstraction and mapping, where the abstraction phase follows specific rules, adapting principles from prior work by Guizzardi et al. [10]. Notably, relators are abstracted based on their properties, while non-sortals are abstracted by merging attributes into sortal classes for clarity. Specializing sortals can be lifted to their super class and phases and subkind partitions can be flattened to an enumeration which becomes an attribute of the general class. In the mapping phase, classes are transformed into schema objects with specific considerations for datatype and enumeration stereotypes. Relations are transformed into attributes with cardinalities, reflecting the essence of the relationship in the resulting API. Response standardization is achieved using JSON:API, generating examples through recursive schema traversal.

While the current research focuses on specific elements of OntoUML, future work may include all constructs to be transformed. It could also present a more dynamic behaviour, such as naming conventions, abstraction methods or which endpoints to use, that the user could pick. A goal may be to include a user interface or facilitate real-time updates.

Overall, this research lays the groundwork for bridging the gap between OntoUML ontology modelling and REST API development, offering a systematic and practical approach to transforming complex ontological models into functional and well-structured API representations.

REFERENCES

[1] 2020. https://dev.realestatecore.io/OWL2OAS/ OWL 2 OAS website, last accessed: 23 January 2024.

[2] 2024. https://www.openapis.org/ The OpenAPI website, last accessed: 23 January 2024.

[3] Chunlei Wu Trish Whetzel Ruben Verborgh Paul Avillach Gabor Korodi Raymond Terryn Kathleen Jagodnik Pedro Assis Amrapali Zaveri, Shima Dastgheib and Michel Dumontier. 2017. smartAPI: Towards a More Intelligent Network of Web APIs. (2017), 154–169. https://doi.org/10.1007/978-3-319-58451-5_11

[4] Alexander Egyed. 2002. Automated Abstraction of Class Diagrams. *ACM Trans. Softw. Eng. Methodol.* 11, 4 (oct 2002), 449–491. https://doi.org/10.1145/606612.606616

[5] Paola Espinoza-Arias, Daniel Garijo, and Oscar Corcho. 2021. Crossing the chasm between ontology engineering and application development: A survey. *Journal of Web Semantics* 70 (2021), 100655. https://doi.org/10.1016/j.websem.2021.100655

[6] Daniel Garijo and Maximiliano Osorio. 2020. OBA: An Ontology-Based Framework for Creating REST APIs for Knowledge Graphs. *CoRR* abs/2007.09206 (2020). arXiv:2007.09206 https://arxiv.org/abs/2007.09206

[7] Gustavo L. Guidoni, João Paulo A. Almeida, and Giancarlo Guizzardi. 2022. Preserving conceptual model semantics in the forward engineering of relational schemas. *Frontiers in Computer Science* 4 (2022). https://doi.org/10.3389/fcomp.2022.1020168

[8] Giancarlo Guizzardi. 2005. *Ontological Foundations for Structural Conceptual Models.* Ph. D. Dissertation.

[9] Giancarlo Guizzardi, Alex Pinheiro das Graças, and Renata S. S. Guizzardi. 2011. Design Patterns and Inductive Modeling Rules to Support the Construction of Ontologically Well-Founded Conceptual Models in OntoUML. In *Advanced Information Systems Engineering Workshops*, Camille Salinesi and Oscar Pastor (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 402–413.

[10] Giancarlo Guizzardi, Guylerme Figueiredo, Maria M. Hedblom, and Geert Poels. 2019. Ontology-Based Model Abstraction. In *2019 13th International Conference on Research Challenges in Information Science (RCIS).* 1–13. https://doi.org/10.1109/RCIS.2019.8876971

[11] Riichiro Mizoguchi and Stefano Borgo. 2021. The Role of the Systemic View in Foundational Ontologies. In *Joint Ontology Workshops.* https://api.semanticscholar.org/CorpusID:240005419

[12] Daniel Garijo Oscar Corcho Paola Espinoza-Arias. 2020. Mapping the Web Ontology Language to the OpenAPI Specification. (2020), 117–127. https://doi.org/10.1007/978-3-030-65847-2_11

[13] Daniele Porello and Giancarlo Guizzardi. 2018. Towards an Ontological Modelling of Preference Relations. In *AI*IA 2018 – Advances in Artificial Intelligence*, Chiara Ghidini, Bernardo Magnini, Andrea Passerini, and Paolo Traverso (Eds.). Springer International Publishing, Cham, 152–165.

[14] Elena Romanenko, Diego Calvanese, and Giancarlo Guizzardi. 2022. Abstracting Ontology-Driven Conceptual Models: Objects, Aspects, Events, and Their Parts. In *Research Challenges in Information Science*, Renata Guizzardi, Jolita Ralyté, and Xavier Franch (Eds.). Springer International Publishing, Cham, 372–388.

[15] Zdeněk Rybola and Robert Pergl. 2016. Towards OntoUML for Software Engineering: Transformation of Anti-rigid Sortal Types into Relational Databases. In *Model and Data Engineering*, Ladjel Bellatreche, Óscar Pastor, Jesús M. Almendros Jiménez, and Yamine Aït-Ameur (Eds.). Springer International Publishing, Cham, 1–15.

[16] Michael Verdonck and Frederik Gailly. 2016. Insights on the Use and Application of Ontology and Conceptual Modeling Languages in Ontology-Driven Conceptual Modeling. In *Conceptual Modeling*, Isabelle Comyn-Wattiau, Katsumi Tanaka, Il-Yeol Song, Shuichiro Yamamoto, and Motoshi Saeki (Eds.). Springer International Publishing, Cham, 83–97.

[17] Lan Yang, Kathryn Cormican, and Ming Yu. 2019. Ontology-based systems engineering: A state-of-the-art review. *Computers in Industry* 111 (2019), 148–171. https://doi.org/10.1016/j.compind.2019.05.003

[18] JUAN YE, LORCAN COYLE, SIMON DOBSON, and PADDY NIXON. 2007. Ontology-based models in pervasive computing systems. *The Knowledge Engineering Review* 22, 4 (2007), 315–347. https://doi.org/10.1017/S0269888907001208