FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Štěpán Stenchlák

Model-driven approach for data schema definitions modeling

Department of Software Engineering

Supervisor of the master thesis:	doc. Mgr. Martin Nečaský, Ph.D.
Study programme:	Computer Science - Software and Data Engineering (N0613A140015)
Study branch:	Computer Science - Software and Data Engineering

Prague 2022

This thesis was made possible thanks to numerous people who helped me or motivated me during the journey.

On the one side, I would like to thank my family for the financial support and for encouraging me to continue educating myself. Thanks to my girlfriend Kateřina for all the emotional support despite the long nights spent on this project.

I want to express my gratitude to my supervisor, doc. Mgr. Martin Nečaský, Ph.D., for guiding me through this and other projects during my study and helping me find the right branch of informatics I enjoy exploring. I also value the help from my consultants Mgr. Petr Škoda, Ph.D. and RNDr. Jakub Klímek, Ph.D.

Finally, I want to thank my friend Jáchym Bártík for the encouragement and the long-lasting friendly rivalry, which will continue in our Ph.D. study.

Title: Model-driven approach for data schema definitions modeling

Author: Bc. Štěpán Stenchlák

Department: Department of Software Engineering

Supervisor: doc. Mgr. Martin Nečaský, Ph.D., Department of Software Engineering

Abstract: This work analyzes, formalizes, and implements a framework for multi-level conceptual modeling of various serialization formats based on Model-Driven Architecture and previously developed tools XCase and eXolutio. It enables users to model their schema from a conceptual model in one general form from which multiple schema formats can be derived alongside documentation and transformation scripts. The thesis introduces base formalisms and findings and analyzes advanced requirements for the following work in this area, such as evolution and inheritance of schemas.

The primary use case of the tool is modeling formal open standards for publishing open data for the government and public institutions of the Czech Republic. Nevertheless, the intent is to make the tool for general schema modeling.

Keywords: schema definition data modeling ontologies Model-Driven Architecture

Contents

Introduction

During the software development process, we may come to a point where splitting a large monolithic codebase into smaller, well-defined modules is necessary to maintain the growth of our software. Using existing systems and connecting them is also a viable approach to building software. [13] In both cases, we end up with many applications and services that work as one.

This approach reduces complexity and demands on software developers as each part can be maintained, deployed, and tested separately. Each developer team must know only the portion they maintain and the nearest surrounding. The surrounding is then defined by a set of rules - an agreement specifying the data which flows between the systems.

Those rules must be created, documented, and kept up-to-date, which can be a long, error-prone task. The result of the process is usually a set of data schemas and documentation for developers. Especially the schemas need to be designed carefully to be, if possible, consistent in format and naming.

Data schemas are computer-readable documents that define the format of the data. They specify how data are structured and how individual properties are named with their possible values. The schemas are used to validate data produced by the application to ensure its correct format and can be used by humans to understand the desired format.

As an example, consider a company selling and distributing its own goods. The goods are stored in warehouses and then shipped to customers. For the shipping process, the warehouse workers need to know the properties of the items they require to send. Similarly, customers need to know the properties of items they are buying. This scenario is denoted in Figure 1.

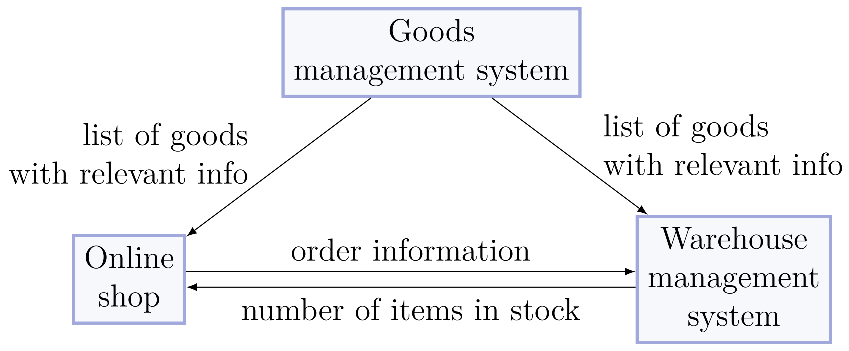

Goods management system
list of goods with relevant info
list of goods with relevant info
Online shop
order information
number of items in stock
Warehouse management system

Figure 1: Example of the company architecture. The nodes are individual systems that communicate with each other.

To describe the data that are being sent between the goods management system and both the online shop and the warehouse, we need to make two "agreements." Although both describe goods, the warehouse may require different properties than the shop. The *name* and *weight* of the items are essential for both systems, but the shop also requires the *price*, contrary to the warehouse, which needs *storing requirements*.

Furthermore, these properties are also reflected in a database schema design of the individual modules. The goods management has a database of goods, whether warehouse management only stores the position of individual goods in the warehouse and the information about warehouses, which is sent to the shop.

Designing the schemas together with supporting documentation by hand is a tedious task, and we may find several obstacles during the process, especially when the schema design is not made carefully:

1. We need to describe the same thing for every schema that uses it. Hence doing the **same work multiple times**. *From the example above, the name and weight of the item are used in two different schemas.*

2. There is a high chance of **inconsistency**. We may name the semantically identical things differently, which may confuse software engineers who came in touch with multiple system parts. *For example, the "name" of the item sold can be interchanged with "label" or "title" in another schema.*

3. It is **hard to introduce changes** as we need to address all affected areas and modify them. If the change is made incrementally, we may lose the context of what is outdated and what is new.

In addition to these obstacles, it is hard to provide supporting documentation, diagrams, and examples.

JSON is an example of a popular format that is widely used to exchange data between the server and the web page as it is natively supported by the web browser. To specify the structure, we can use JSON schema (see Figure 2), a document written in JSON format.

Diagrams are images that visually explain the domain or the data structure. Figures 2.6 and 4.2 can be considered as examples of diagrams that may help to better understand the structure. Diagrams are especially helpful in showing relations between various things that are used in schemas. For example, the order is made by a user, consisting of goods stored in warehouses.

Examples can be provided on two levels. Programmers may appreciate sample data that are valid against the schema and can be used to test their applications. Figure 2.11 is an example, although it would be better to provide more than two items. Examples can be provided for individual things as well. Someone may not be sure how long the ideal description is or what is the naming convention of items.

The **documentation** would describe each property of the schema in more detail in human-readable form. For instance, documentation may be a website with multiple pages where the schema, diagrams, and examples are included, and all properties are described in formatted text. Individual things may be interlinked to ease the discovery of related things.

Future modifications to the system or changes in user requirements may enforce altering the schemas. Formally, the process of changing schemas is called **evolution**. The evolution is complex, as there can already be existing data that conform to the changed schemas. In that scenario, properly implementing a change in a user requirement requires modifying affected schemas, documentation, and all the data (in case the data are stored in the given format).

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Item",
  "description": "Single item that can be sold in the store.",
  "type": "object",
  "required": [],
  "properties": {
    "name": {
      "type": "string",
      "title": "Item full name",
      "description": "Descriptive, short text, usually provided
                      by the manufacturer."
    },
    "price": {
      "type": "object",
      "title": "price",
      ...
    }
  }
}
```

Figure 2: Example of JSON Schema that may describe the data being sent between the Goods management system and the online shop.

As an example, suppose that the address provided by a customer to deliver the goods is represented in multiple parts such as *street*, *city*, *zip code*, etc. We may decide that it would be more reasonable to have everything in one field as some parts of the address may be missing or not granular enough. In this case, the evolution would mean changing all the schemas, documentation, and examples to reflect the new structure and possibly creating transformation scripts to convert old data to the new format.

Ontology

The whole process of designing schemas needs to be perceived on two levels. (i) On a technical level, where a user creates the schemas and describes the rules, (ii) and on a conceptual level, where the things and the relations between them are defined. The latter is called an **ontology**.

An ontology describes and names relations between the concepts from real life without the technical details. Concepts are, for example, *order*, *customer*, or *goods* in our case. Relations then specify, for example, that *the order belongs to a customer* and *consists of goods.*

Having an ontology properly defined is a step in the right direction as it gives us a template for schema modeling. All things we may need to employ are described in the ontology, ensuring consistency across schemas and helping to avoid mistakes.

With the data on the web [2] trend in the last few years, even ontologies are becoming accessible publicly. Popular formats include RDFS (or RDF Schema) [1], OWL[2], UFO [7], schema.org, and Wikidata[3]. For example, schema.org is a proprietary format describing valuable data for search engines, such as events, organizations, or places.

Using pre-defined ontologies in a semi-automatic way of defining schemas is beneficiary because a schema designer may focus entirely on the schema structure and not on the domain semantics.

An alternative problem

In the section above, we have introduced a problem behind data modeling during software development as it is hard and time-consuming to design schemas properly. Nevertheless, schema design may be used in a broader to create recommendations for publishing various data in a unified format.

This task is usually undertaken by the state administration to ensure interoperability between various state and private organizations. The European Union enforces this in the directive 2019/1024 of the European Parliament, that data of public institutions shall be published as open data on the Internet in all formats it was created and, if possible, in a machine-readable format.

Open data Open data is a term for data published on the web without any restrictions on use. This means that anyone can use, modify and distribute the data for any reason, including commercial use.

The definition of open data is very loose but can be further specified by a 5-star scheme designed by Sir Tim Berners-Lee. Each star adds a restriction up until the fifth star describing the Linked Open Data.

1 ★ Data are published on the web and can be used freely.
2 ★ Data are structured and in a machine-readable format.
3 ★ The format is not proprietary; hence anyone can open them.
4 ★ Data uses RDF and SPARQL standards from W3C.
5 ★ Data are linked to other data creating a network of data.

The act then specifies **FOSes**[4] *(Formal Open Standard)* as recommendations for publishing selected categories of data, such as information about *Tourist destinations* and *Sports centers*. The purpose of these documents is to standardize how these data are published, usually by defining JSON and XML schemas along with textual documentation.

The process of designing those recommendations is comparable to designing schemas for a software system mentioned above - the designer needs to create schemas and documentation for them with examples and diagrams.

[1]https://www.w3.org/TR/rdf-schema/

[2]https://www.w3.org/OWL/

[3]https://www.wikidata.org/

[4]https://eur-lex.europa.eu/legal-content/en/TXT/?uri=CELEX%3A32019L1024

Besides the different use cases, the process of designing the specification is slightly different, as we need to provide multiple formats to not restrict the publishers by using specific technology, compared to the previous example, where a single format of data is sufficient. This brings us to several other challenges that are usually related to ease of the publications of the data:

1. As there are multiple different formats, we need to design tools that can convert data between those formats. Because the conversion depends strictly on the schema, it is related directly to the schema modeling.

2. To ensure the publishers that data are published correctly, we shall provide them with a platform for testing. This is also significantly related to schema modeling, as the purpose of the platform is to visualize the provided data to show that everything works.

Focus of the work

This thesis analyzes and implements a tool Dataspecer [19], and formally defines and analyzes its internal framework, which follows the previous research [10, 11, 14–16] in the XML data modeling area and extends it to support new requirements.

The purpose of the tool is to ease the process of creation and management of data specifications, such as XSD, JSON Schema, and CSV Schema, and the creation of supplementary documents. The tool shall help users model schemas by providing relations and entities from the chosen ontology and letting them focus on the modeling part only.

Supplementary documents are automatically generated files from the modeled schemas, such as the documentation, examples, diagrams, transformation scripts, and others.

The tool is part of the larger ecosystem of tools, libraries, and frameworks for data modeling, which is being developed by the same authors.

Due to the complexity of the topic, this thesis does not attempt to cover and implement all features, as some of them will be kept for the authors' future work.

The purpose of this thesis is to provide basic formalisms and findings for the next work in this area.

The rest of the thesis is organized as follows. The following chapter 1 introduces the previous research as the common ground this work will follow - precisely the model-driven approach for data modeling and evolution of XML documents. The succeeding chapter 2 analyzes new requirements for the tool and proposes a solution to some of them. The rest in chapter 3 is only briefly analyzed to set a direction for future development. The solution is then formalized in chapter 4. The next chapter 5 identifies key implementation decisions and describes them. Then chapter 6 examines related work in the area of data modeling. The last chapter 7 evaluates the tool in the context of use in modeling FOSes for the government of the Czech Republic.

1. Previous research

This short chapter introduces the core concepts of the previous research carried out by the *XML and Web Engineering Research Group* (XRG) at the Faculty of Mathematics and Physics of the Charles University from 2012 to 2015. Those concepts will be used for further analysis in the following chapters.

In 2012 XRG formalized a novel approach[1] to modeling XML schemas [16] for a particular domain ontology by integrating Model Driven Development [8] (specifically Model Driven Architecture) techniques to separate a conceptual model describing the domain ontology and a structural model which described the concrete XML schema.

Model-Driven Architecture MDD is a software engineering technique that abstracts software development into several levels (models) to allow flexibility by separating the business domain from platform decisions. MDA [17, 18] introduced by Object Management Group (OMG) is a specialization of MDD focusing on the automation of the development of software systems.

MDA introduces four models. The topmost level CIM (Computation Independent Model) expresses the business logic and has no formal representation, as it is only a concept, hence the name computation independent. As a second level, PIM (Platform Independent Model) models the business logic in UML. It does not specify a concrete platform or technology but only the concepts in a formalized way. PSM (Platform Specific Model) reflects the formalized concepts from PIM in a platform-specific environment, such as in XML Schema, C# or Java code, or a database schema.

The key concept is a transformation as the process of mapping the upper layer to the lower one. The transformation from CIM to PIM must be done by hand as CIM does not formally exist. More interesting is PIM to PSM transformation, as it can be automated. The transformation keeps the mapping to preserve the semantics between the models. The last level is Implementation Specific and only represents different implementations of PSM.

XRG used PIM and PSM in their architecture. PIM represented the domain ontology in UML-like notation[2], as it is independent of the platform as XML. PSM then defined the given schema in their own designed grammar, which was translated into a final schema, such as XSD (which formally corresponds to the Implementation Specific level).

The major benefit of the strategy presented is a shared conceptual model between various XML schemas, as other works at that time had a single conceptual model for every schema. This was not practical as usually multiple schemas are applied in a single software. Authors have also formalized the model and have proven that their approach is correct. That means that (i) every conceptual schema models XML schema, (ii) their translation algorithm from the internal

[1]The work builds on previous work of the same authors, who introduced the XSEM model [14], which was later a subject of their study.

[2]Formal definition of PIM and PSM levels is in chapter 4.

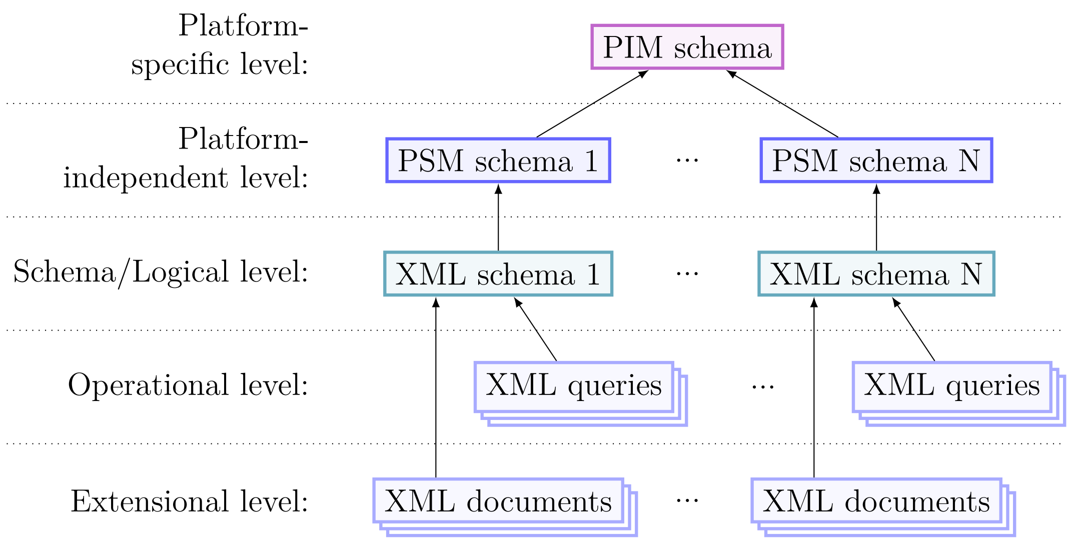


Figure 1.1: The five-level framework proposed by XRG in [16]. A shared PIM layer with a conceptual model is used by multiple PSMs, where schemas are defined in their own grammar. The grammar is then translated into schemas that conform to XML documents in the last level.

model to schema respects introduced rules and is reversible, and (iii) their normalization and optimization algorithms produce semantically same schema.

Their primary use case was creating schemas for the government, such as the National Register for Public Procurement (NRPP) information system for publishing public contracts.

Modeling To create a schema, a user must first create an ontology of desired entities as PIM. Then, multiple schemas can be made as PSM trees by selecting a schema root and then adding entities and setting their properties. The resulting XML schemas can then be exported from the application.

Evolution of schemas The focus of XRG was also directed to the evolution [15] of their proposed model to minimize the work of the data designer. As already stated in the introduction of the thesis, changes may be inevitable (either from the user requirements or the surrounding environment) in large and complex systems, and propagating even a tiny change from the domain ontology to all affected schemas is time-consuming and error-prone.

They proposed, formalized, and later implemented a solution in restricting the changes in PIM and PSM models to only atomic operations - simple changes in the model, such as *creating a new class*, *updating a name of the association*, or *removing an attribute*. Those operations are not intended to be used by the user directly but are simple enough to be formally defined and mapped to the corresponding operations in the level below. The proposed mapping is then used to propagate changes in the model to the schema level, more precisely from PSM to PIM, which is then translated to the schema level. They implemented only top-down propagation of changes as the propagation from XML documents is usually not meaningful when modeling from ontology (but theoretically possible).

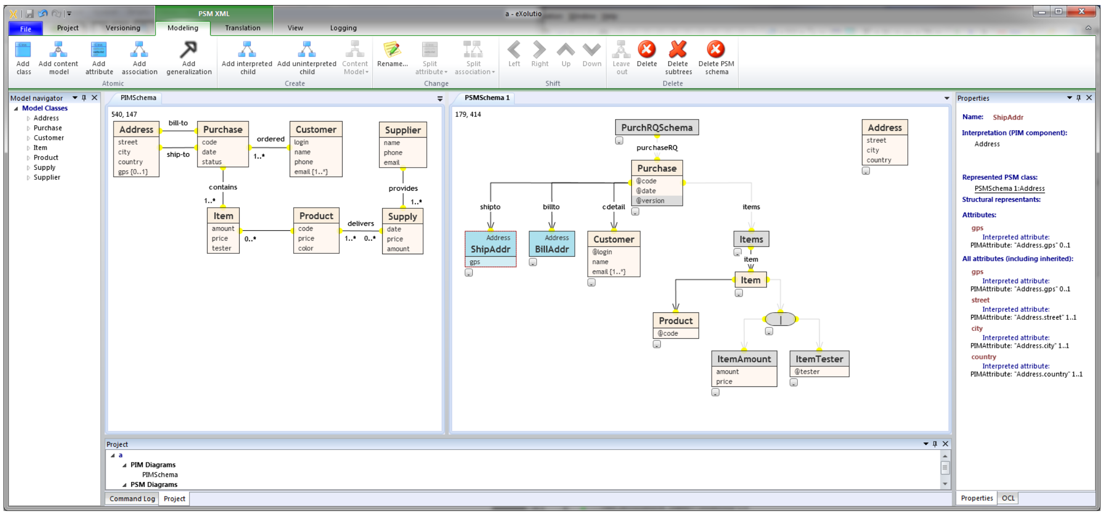


Figure 1.2: Preview of the eXolutio tool. The left panel shows the PIM model as a UML class diagram, while the right panel represents a PSM schema in a tree-like structure that can be converted to an XML schema.

Implementation Their result was implemented in two tools *XCase* [10] and *eXolutio* [11]. The former one was simpler, focused only on modeling. The latter then supported schema evolution as described in the previous section. Tools let users define the ontology from which the schemas and operations were derived for XML documents.

The tools were designed only for XML, hence are not usable for other languages, such as JSON, which is very popular at the time of writing this thesis, as being used by many server-centered applications to communicate with the server through the REST API [6], for no-SQL databases[3],[4] and more. Nevertheless, we can use their findings and generalize the model for other formats.

They focused on the correctness and completeness of the model, which in some cases may be a limitation, such as more complex implementation and processing of the model not to allow a user to create a schema that is not valid.

Lastly, the tools didn't support sharing and collaboration, as this was not considered standard practice in the industry at that time.

[3]https://www.mongodb.com/
[4]https://rethinkdb.com/

2. Requirement analysis

This chapter summarizes the expectations for the application in the form of requirements. The requirements are analyzed, and in the following chapter, the solution is proposed with a focus on a formal description of the framework.

As **stakeholders**, we would consider data analysts, programmers, or at least individuals interested in the area of data modeling, since the typical use case of the application is (i) to design schemas for a large system of interconnected subsystems or modules or (ii) to design a recommendation for publishing data. Both these use cases are described in the introduction of this chapter.

Because of the stakeholders' knowledge in the area of data modeling, we may keep the UI of the application more technical as the intent of all operations may be intuitive for them. Nevertheless, the basic functionality does not require advanced knowledge in the abovementioned fields, so we propose an "expert mode." The user will be asked whether they feel like an expert in the area, which would make available more advanced application features while keeping the UI simple for those interested in the basics of data modeling.

2.1 General schema

Requirement 1. A user shall be able to easily derive a **general schema** structure from the existing ontologies and then translate the structure into different known schema languages, such as JSON Schema, XSD, and CSVW Schema and it shall be possible to add support for others easily.

The basic idea behind this requirement was already explained in the Introduction. From an ontology specifying the relations between things from a real world, it should be easily possible to select things and relations between them that describe a schema. The schema then defines a structure of data that represents those things from an ontology.

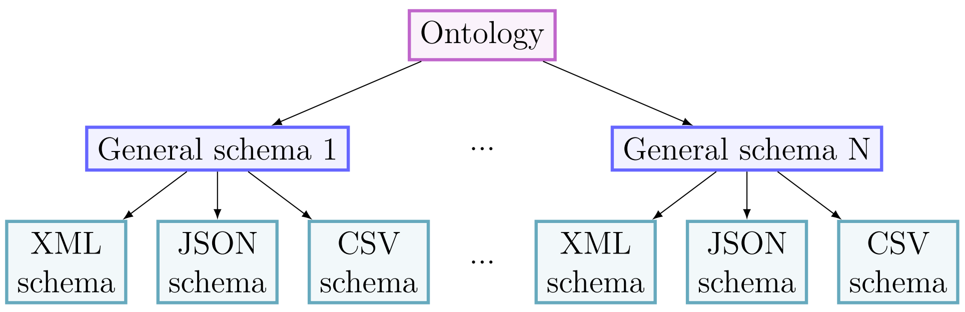


Figure 2.1: Diagram showing the core workflow behind the data modeling from an ontology. Users can create general schemas (blue rectangles) from the ontology. From those schemas, the tool creates data schemas in known formats, such as XSD, CSV Schema, or JSON schema.

We aim to design a model for a **general schema** that can describe most of the serialization data formats. This model will be used as a mapping from the ontology to the desired schema. The model must be robust enough to support different formats, as we want to use the same general schema for all of them, which corresponds to MDD as an abstraction layer.

There are many formats for data exchange, the most famous being JSON, XML, CSV/TSV, and RDF. Data formats can be categorized into the following categories based on the structure model:

- The **hierarchical model** stores data in a tree-like structure, having one root class with properties that may recursively contain other classes. It has been one of the most common models for data serialization in the past few decades, as it is easy to understand and interpret and is suitable for most types of data. XML and JSON are examples of formats that use this model.
- **Relational model** uses a set of tables to store data. Each table represents a sequence of similar things, each on one row with columns as properties. Rows may point to rows in other tables to link data. The relational model is also famous for its simplicity in CSV and TSV files, which can be easily parsed. The relational model is also used in relational databases (the databases that use SQL query language).
- **Graph model** represents data in a general graph structure with nodes and edges. RDF (Resource Description Framework) became a popular format using the graph model, where nodes usually represent things or literal values, and edges connect them as properties.

As our primary intent is to support JSON and XML, we will use the first type of model to represent the data in our general format. The translation from that format to individual schemas in the hierarchical model would be implicit (and will be described later in the text).

Supporting translation from the general schema, which is in the hierarchical model, to the formats in the relational and graph models should be possible in a limited way[1], which is sufficient and follows the requirement to have one general schema.

The graph model is not even necessary to generate as we use the ontology that is already in the graph model; hence, we can use the ontology directly as the schema to validate our data.

2.1.1 Analysis of the formats

We will analyze the standard formats to properly design a user interface for the schema modeling and the underlying general schema model capable of describing those formats.

JSON (JavaScript Object Notation) is a simple format with two complex data types: objects and arrays. The objects represent data in key-value pairs with values that can have any type, including other objects and arrays. Arrays then

[1]That means we may not be able to reverse translation from a specific schema to the general schema, or it may not be possible to use the full power of the given specific schema. However, this is not important to us, as our target is the support of basic use cases.

represent lists, and both arrays as objects may be in the root of the document tree. Semantically, objects represent things with their values as properties.

XML (Extensible Markup Language) is similar to JSON since both formats are hierarchical. XML tags wrap parts of the document representing either things or properties of things and can be nested similarly to the JSON format. In contrast to JSON, XML tags can have attributes.

```
{
  "id": 3758,
  "title": "Chair",
  "variants": [
    {
      "title": "Black",
      "price": 200,
      "color": "black"
    },
    {
      "title": "White",
      "price": 200,
      "color": "white"
    }
  ]
}
```

(a) JSON document - braces `{}` wraps object and brackets `[]` wraps array

```
<Good id="3758">
  <title>Chair</title>
  <Variant>
    <title>Black</title>
    <price>200</price>
    <color>black</color>
  </Variant>
  <Variant>
    <title>White</title>
    <price>200</price>
    <color>white</color>
  </Variant>
</Good>
```

(b) XML document - `<Good>` tag serves as a class wrapper, whether `<title>` has a property meaning

Figure 2.2: Comparison of JSON and XML format both showing data about the same chair.

As seen in Figure 2.2, the XML format is more complex, as it supports tag attributes (see `id="3758"` attribute), and arrays can be written in two distinct ways. We can place elements of the array directly in the parent container, as we can see with the `<Variant>` tag, or we can wrap them into another container for clarity (for example, into `<variants>` tag).

JSON Schema is a JSON document that describes the data structure we can expect from other JSON documents. For this part of the thesis, it is sufficient to know that the schema defines which root object we can expect and a set of allowed properties and their types for each object.

Suppose that we have chosen a structure very similar to JSON Schema to be our general structure format. We are interested only in how it describes the document's structure. Because JSON is simpler than XML, we can use our model to describe only some XML documents, as we are missing constructs that would describe advanced XML features.

For example, the object property x with primitive value y would represent an XML tag `<x>y</x>`; if `y` is an object, we will apply this rule recursively. The object property x with an array of y_i would represent multiple XML tags `<x>`y_1`</x><x>`y_2`</x>...<x>`y_n`</x>`. Finally, we will start with the root tag, which was `<Good>` in our case.

To describe and distinguish between more advanced XML features, we would need to add XML-specific options to our model, such as:

1. For every object property with a primitive value, there should be an option that the given property becomes an attribute of the parent tag. For example, the `id` property of the chair may be either the attribute `id="3758"` of the parent or the full tag `<id>3758</id>` inside the parent.

2. For every array property, there should be an option that the given list of tags will be wrapped.[2]

3. XML, compared to JSON, recognizes the order of the elements in the document. This means that we may decide whether we want to enforce the specific order or not, which can also be fixed by another option in the parent.

Comparing the structure of JSON and XML once again, we can let a user use the JSON Schema-like structure with optional annotations for advanced XML features. This allows us to have a simple model which is easy to understand and operate and can be annotated by other options for specific languages, as we have shown for XML.

CSV (Comma-Separated Values) or TSV stores the data in tables. Unfortunately, this means that the structure is entirely different from the case of JSON and XML. Because having a separate schema would cause complications against other requirements, we will analyze whether it is possible to translate our general structure format from a hierarchical model to a relational one.

In the general case, there are existing approaches [3, 9] to map the hierarchical model to relational. Therefore, we will only show a brief example. Suppose that our general structure format contains objects, properties, and arrays. From each object type, we will create a table with columns as properties. Each table must have a primary key so that the tables can be linked together. If the schema contains an array, we will link children to the parent table; thus, the array properties will not have a column.

id	title
3758	Chair

good-id	title	price	color
3758	Black	200	black
3758	White	200	white

Figure 2.3: Document of two CSV tables representing the same data as in Figure 2.2. The left table contains the root.

Because all tables represent arrays, we cannot formally convert the schema with an object in the root. We simply suppressed this in Figure 2.3 by wrapping the schema root into the array.

To support CSV documents containing unrelated data (and possibly other unrelated data in the relational model), specifically CSV tables, that do not reference each other, we may need to have a schema with multiple roots. Multiple

[2]This may not be necessary as we can add the wrapper class to the general schema by ourselves. This, however, is not the correct solution as the meaning of the wrapper class is not semantical but rather syntactical and only specific for XML.

root schemas may be helpful in some advanced data-modeling problems. We will keep the question behind this open as there are not enough use cases right now.

Although we have not dealt with advanced cases, the model is robust enough for most of them.

2.1.2 Designing the model

So far, we have shown that a JSON Schema-like model with format-specific annotations is sufficient for describing the structure of JSON, XML, and CSV documents. In general, we cannot have too strict requirements on the model, as some other formats may not require all the information or might be too simple. This pushes us to define the schema in the most elementary way.

We will allow only classes to be a root of schemas and instead add an option that the root can be an array. This simplifies the work with the model, as we may always expect a class.

Classes then have an ordered list of properties. This is different from JSON, where properties have no order. A property may be an attribute or association. An attribute has a primitive type, such as a string or a number. Association is a property with another class. Because we have forbidden using arrays in the root, we omit them entirely as an array of primitive values and classes can be achieved by the cardinality of attributes and associations, respectively. Cardinality is an interval specifying how many values a property can have. 1..1 is for required properties, 0..1 for optional, and $0..*$ for arrays.

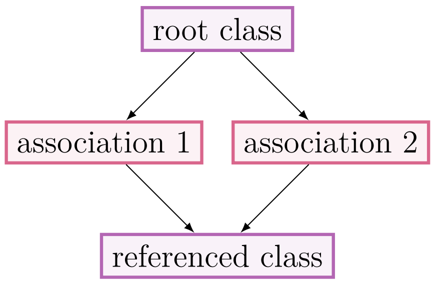
We can use two different approaches to visualize the model's hierarchical structure. The previous tools *XCase* and *eXolutio* used graph visualization, where nodes were used to show classes and edges to show associations. An alternative approach is to use a textual "bullet-list" representation, as the model is *usually* a tree.

The latter approach is easier to understand, as the final product is a schema for documents that has a similar structure as the representation. It is easier to implement, more compact in size on the screen, and easier to work with on smaller devices. Also, the order of the properties is more intuitive, and we can use more styling options for advanced constructs.[3] However, in the general case, users may benefit from the graph view if the schema refers to another schema (see Requirement 5) multiple times or contains cycles because this can be easily denoted in the graphical interface (see Figure 2.4).

Because the primary use case is to generate simple or moderately advanced schemas, the textual approach is preferred. Nevertheless, the graph view might be implemented in the future.

As shown in Figure 2.4, the schema may be represented as a "bullet list" where each class, association, or attribute is on a separate line. Classes have a list of properties under the class name. Associations point directly to other classes and, therefore, can be merged with the class name on a single line. Other attributes, including format-specific, will be on the line next to the item name.

[3]So far, we have described only a basic schema structure. See other requirements for advanced constructs.



(a) Graphical representation

```
root class
  - association 1 to
      referenced class
  - association 2 to
      referenced class
```

(b) Hierarchical representation

Figure 2.4: Figure showing a schema referencing the same subschema twice, essentially creating a cycle in an unoriented graph. Two different representations are shown - graph and hierarchical. The former shows that both associations refer to the same subschema, which later representation cannot show explicitly.

```
class Good
  - attribute id[1..1]: string
  - attribute title[1..1]: string
  - association variants[0..*]: Variant
    - attribute title[1..1]: string
    - attribute price[1..1]: number
    - attribute color[1..1]: string
```

Figure 2.5: Proposition for how the general schema may be represented for the example that validates data with the chair.

It shall be possible to change the order of the properties by dragging them. Options for given items shall be available next to them. Attributes and associations shall be distinguished both by color and supporting graphics. More advanced constructs may have unique styling options to provide more information if necessary.

2.2 Ontology

Requirement 2. As many ontologies are located on the web in formats such as OWL (Web Ontology Language), RDFs (RDF Schema), and UFO (Unified Foundational Ontology), the application shall support reading them.

It may seem that designing the ontology directly in the tool is beneficial because a user does not need to use other tools, and the application may continuously build the schema, which is feedback to the user. This approach was used in tools *XCase* and *eXolutio*, as seen in Figure 1.2 in the left panel. However, it has the following drawbacks:

1. Designing an ontology is a well-defined problem. There are many great and time-proven tools we could not cope with.

2. Even if the intent is to use the ontology only to generate the schemas, it may be worthy of publishing it anyways as others may benefit from it.

3. It is better to split a complex problem into smaller ones.

On the other hand, not having direct access to the ontology, as it will be on the web, has the following impacts:

1. The ontology may **not always be available**. Unavailability should not prevent us from generating the schemas and making minor changes to them if those changes are not directly related to exploring the ontology.

2. The concepts in the ontology may **point to another ontology** according to the Linked Data principles.

For the reasons mentioned above, the preferred workflow is to design the ontology separately in the external tool, publish it on the Web, and then model the schema in the application. There is Requirement 8 later in the text specifying that a user can make modifications directly in the application. This is not inconsistent with the statements, as it deals with minor changes instead of defining a complete ontology.

The term ontology has already been defined in the introductory chapter, and we will formally define it in the next chapter.

It shall be easy to implement support for other types of ontologies, and all of them shall be linkable according to the LD principles.

2.2.1 Format of the ontology

In the above requirement, several different formats were proposed for the ontology. This section will analyze the minimal requirements for any ontology format and how we will treat additional information in them. Because the core goal is to design schemas, we will start with a model proposed in Requirement 1. The schema consists of classes and their properties. A class corresponds to a thing from real life. An attribute is a literal that belongs to the given class only. On the other hand, an association is a link between two (not necessarily different) classes. From this point of view, the association is an independent entity.

Associations are usually oriented, and some ontologies may specify a title and a description for a reverse direction. For example, in RDFS, the association (or property in the RDFS terminology) is an entity of type `rdf:Property` having domain and range classes and a title and description. Therefore, it only describes the forward direction. We can, of course, create a property in the other direction as well, but there would not be a connection between a forward and a reverse one. Hence, we would not know that those are semantically the same properties. UFO (Unified Foundational Ontology), as an example of a more complex ontology, introduces relators. Relators are relationships between two or more things connected by mediation. The mediation can be described, giving us a way to describe both directions differently.

Although the latter approach is more complex, using simple concepts for associations may be disadvantageous for the aforementioned reason. Therefore, we will follow the pattern of UFO, and any other simpler ontologies, such as RDFS, will not have the reverse direction described.

An ontology in the context of this requirement is a set of classes that have attributes. The associations then connect two classes together. The connected classes may not have the same ontology.

As some formats specify the ontology more complexly, the application may use the additional information to design the schema better. This statement is more defined in the following chapters.

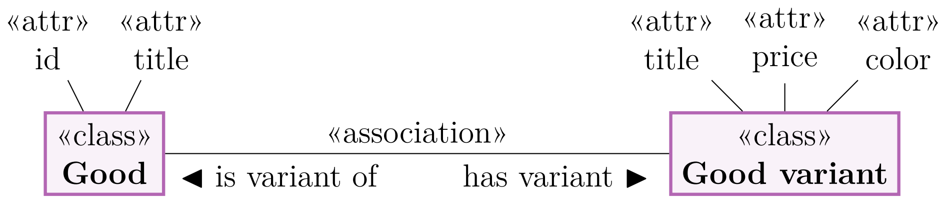


Figure 2.6: Schematic diagram of an ontology that could be used for the schema from Figure 2.5.

2.3 Data modeling analysis

2.3.1 Type coherency

As already mentioned, an ontology is not just a supporting source for the modeling process but rather the only source we can use to create schemas. The schema then represents a mapping to the ontology for further processing.

Because parts of the schema are mapped, we can check whether the attributes and associations belong to the given class. This allows us to check whether the schema is being built correctly and to provide the appropriate help during the modeling based on the type of the classes.

Although the problem may seem trivial, there are advanced scenarios that must be considered.

1. We may want to add additional attributes and associations directly into the schema without a connection to the ontology. This is a schema-modeling problem as we may need, for example, to wrap several properties into an additional object (JSON) or a tag (XML) or add another property because the data we validate contains it.

2. If A is associated with B and we have a schema with the class A having B, then it may be possible to move attributes[4] from A into B. Because for each B, we know to which A it belongs, we do not lose any information during this process.

As an example of the second case, suppose that our ontology has *Goods* and their *Variants*. Variants are colors, sizes, and materials for the given item. Indeed, all variants are made by one manufacturer. Therefore, it makes sense that the

[4]Moving of properties to different classes will be kept as future work. Nevertheless, to cover some use cases, we employ a simpler construct of dematerialization. Association that is dematerialized is removed from the generated result, and all properties from the associated class are moved to its place.

manufacturer attribute would be associated with the *Goods* class, whether the *color* with the *Variants*. This may not be a beneficiary for all data consumers. Therefore, a schema with the *manufacturer* attribute moved into the *Variants* class might be a better solution.

```
{
  "title": "Chair",
  "variants": [
    {
      "price": 200,
      "color": "black",
      "manufacturer": "IKEA"
    }
  ]
}
```

```
{
  "customerId": "12",
  "personal-info": {
    "name": "John",
    "surname": "Doe"
  },
  "contact-info": {
    "address": ...,
    "phone": ...
  }
}
```

Figure 2.7: Examples of data with some attributes moved. The former moves the attribute *manufacturer* from the parent into the other class that has a counterpart in the ontology. The latter takes attributes such as *name* and *address* and wraps them with additional objects that do not correspond to the ontology.

However, this is too complex for the current state of development, but it gives us a chance to think about the problem in a more general way.

2.3.2 Data modeling process

So far, we have only described the desired structure of an ontology and a schema model, but we did not tackle the actual process of how the schema is created.

A user starts by selecting a root of the schema. Schema under the given root would then describe one entity of the given type, or a list of those entities, depending on the later configuration. Because a set of possible root classes is not limited in any way (or, we can say that the root has the most general type, hence can be specialized), the most suitable option is to let the user search for the class by its name, descriptions, or other parameters, depending on the given ontology format.

As soon as the root is placed in the schema, we get a context because the following attributes and associations to other classes depend on the class where the properties are being added. For that, we use a prompt dialog where it is possible to select those properties to be added.

Although we did not enforce that an ontology must support inheritance, most of them do. Therefore, the dialog also allows adding properties of the parent class.

The process of adding properties can be more automated in the future. The tool can propose to automatically add all attributes and associations with nonzero cardinality, as it may be the desired behavior. We can perform this action even recursively to automatically design the whole schema just from the root and select a few options where the recursion shall stop.

In general, there might be cases with hundreds of available entities to add where only some of them are relevant to the current user. This is further supported by the fact that anyone can extend our ontology by introducing their classes associated with ours. To properly understand the relevance of different entities, a user profile from past choices needs to be built. This is an area of recommended systems, with many focusing specifically on model-driven engineering [1], and we will consider them in our future work.

Requirement 3. The application shall create supporting documents for the generated schemas.

The main goal of the tool is to model schemas from a given ontology. Nevertheless, to better understand the generated schema, the documentation, possibly with diagrams and examples, is very beneficial.

A structure of the documentation was already described in the introductory chapter and can be easily derived, as it only describes used concepts that are mapped from the schema.

Regarding examples, for the schema in Figure 2.5, Figures 2.2 and 2.3 could be automatically generated. This would require additional knowledge from the ontology as the application needs to understand that the title should be a buyable item (hence *chair* and not *sitting* for example) and the price should be reasonable to the item's actual price (because it may confuse users when the price would too low or too high).

2.4 Data transformations

Requirement 4. The application shall support generating transformations between different data conforming to supported schemas and RDF representation.

Data transformations were also introduced at the beginning of this thesis as the necessity for the latter use case of interoperability of public institutions. Data transformations are generally used to convert data (not schemas, but data that conform to given schemas) from one schema to another without changing its meaning.

One example may be to convert CSV to a JSON array of objects, where each object represents a row in the CSV. There are plenty of online tools to do this, but they do not understand the context of the data. Because both schemas were derived from one general schema in the tool, we may correctly map columns from CSV to the fields in a JSON object, not necessarily from a single-table CSV schema.

In the context of this tool, transformation means both (i) transformation between different schemas under the same general schema and (ii) between different general schemas, if possible (as we may exploit the knowledge of the mapping to the original ontology). As an example of the second case, we may have two general schemas for the same thing, where one is simpler than the other. For example, suppose the schema from Figure 2.5 and similar with more attributes and associations, possibly with a different order of properties and labels. It is then possible to convert the data from the more complex schema to the simpler one by losing the information. If default values are provided or additional properties are optional, the transformation in the other direction should also be possible.

Regarding the transformation process, there are plenty of ways to transform the data:

1. Data engineers use **Python** with support for many formats using libraries. In this case, the transformation would mean a generated Python script with a predefined interface that takes data from one format and outputs in another. Depending on the use case, the script may be configurable (besides the possibility to configure the generation of transformation itself).

2. There is **XSLT** (Extensible Stylesheet Language Transformations) language to transform between XML documents or from XML to XML-like, plain-text, or CSV documents. XSLT is an XML document that can be executed with an input document by an XSLT processor, producing the resulting document. A disadvantage is that the input document must be in XML format; hence, it cannot be used alone for bidirectional JSON and CSV transformation.

3. There are mapping tools and languages, such as **RML** [4] (RDF Mapping Language), designed explicitly for mapping purposes. RML maps common serialization frameworks, such as XML, CSV, and JSON, to RDF from a set of rules written in RDF. The translation mechanism is similar to the XSLT. Specifically for JSON, there is **JSON-LD**[5] with simple directions to set mapping to RDF. Conversion tools are available in multiple programming languages. There is **CSVW**[6] for CSV as an alternative to the previously mentioned JSON-LD.

Although RML is a ready-to-use solution with support for all three technologies, it requires its own transformation toolchain. This is also valid for JSON-LD and CSVW technologies. On the contrary, XSLT is a well-known technology among people working with XML and is widely supported. Our primary goal is to have transformations that are easy for stakeholders to use in their systems. Therefore, we will implement XSLT for XML while keeping RML for later.

Similar to the translation of a human text, there are two approaches. Either create a transformation for each pair or have one standard format where all the others can be transformed and vice versa. The latter method requires only one transformation for each new format added and is easier to debug, as there is a middle format. Because schemas are built from ontologies whose primary source is RDF, we will exploit this and have RDF as the middle format, which is another format in which we can transform data.

We categorize two types of transformation, lifting and lowering. Lifting is a process of converting semi-structured data such as JSON, XML, or CSV into RDF. Lowering is the opposite process. By combining them, we can achieve a transformation between various formats. That means that even if we want to transform XML to CSV, which would be possible by a single XSLT document for simple structures, we would need to execute two transformations.

[5]`https://json-ld.org/`

[6]`https://csvw.org/`

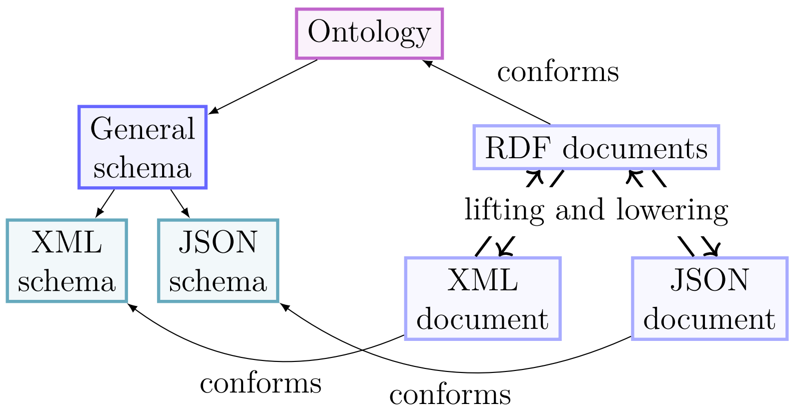


Figure 2.8: Example of data transformation. An XML document that conforms to XML schema may be lifted to RDF representation, which conforms to the ontology. The RDF can then be lowered to another format.

2.5 Data specification

The following requirements urge us to group similar general schemas into a project that we call a **data specification**. Schemas in the data specification may share some configuration or depend on each other, as we specify later. Each schema belongs to exactly one data specification. We will not determine in this thesis which schemas should share a data specification and which should not because there are currently no limitations that would state otherwise. However, this may change in the future as new requirements arise.

Requirement 5. It shall be possible to refer to other schemas to use them as building blocks for larger ones. Schema reference shall be treated as a reference to the resulting schemas and documentation as well.

Referencing other schemas is crucial for advanced use cases where it is essential to split large schemas into smaller blocks that can be published and used separately.

For most schema languages, it should be sufficient to refer to the other schema as is. For example, in JSON, we can use the `$ref` keyword with a path to the referenced schema. On the other hand, data transformations may not always be able to handle this approach. Hence, having a full copy of the schema might be necessary. Referencing a schema would thus require access to all data in its specification.

To avoid problems with tracking references and knowing which data specification needs to be loaded to generate artifacts properly, a user would need to set a given **data specification that is being reused** explicitly. Similarly to the Requirement 2 with the ontology, we do not require that the reused data specification be always available. The application shall work even if the data specification is not available at the moment if the presence of the specification is not required directly, such as for creating a new reference or generating artifacts that depend on it.

Requirement 6. The list of supported schemas, transformations, documents, and other files generated from the general schema shall be easily expandable to adapt the tool to different use cases.

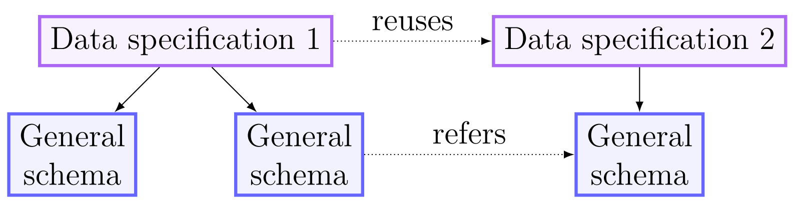


Figure 2.9: Example of reusing of specifications. All schemas from a reused specification become available to refer from local schemas. Only the root of the schema may be referred to.

The generation of schemas is robust enough to be used in every common scenario. Therefore for most users, we do not expect they may want to intervene in the process besides the standard configuration, such as indentation, using of comments, or a default language.

On the other hand, documentation is a very vague concept that neither we have adequately specified. Sometimes simple Markdown documentation may be sufficient, while elsewhere, the user may require a strict format of multiple documents in HTML.

Transformations have similar issues. There are multiple ways and technologies that transform data between different schemas. We have already mentioned transformation through the RDF format, either by RML or custom scripts, such as XSLT for XML. For the more demanding user, it is even possible to create transformation scripts between pairs of different serialization formats, such as between XML and CSV.

We will expose a way the user can register their own generator that can create a set of files in a filesystem from the given schema.

To support the linking of generated files, generators may use others to modify their results, further expand them, or create a link to them. This is specifically useful for documentation, as it should contain links and possibly a copy of generated schema.

2.5.1 Artifacts

There is little difference between generated schemas, data transformations, documentation, and other output files. Based on the general schema and provided configuration, if any, the application shall create a set of files that can either be published on the Web or stored in the file system. All generated files will be denoted as **artifacts** and created by **generators**.

We will distinguish two types of artifacts. (i) **Specification artifacts** do not depend on a concrete schema but use the whole specification. Documentation may be an example of a specification artifact because it generates a single document concerning all schemas. Of course, schema-specific documentation is possible, and it purely depends on user requirements. (ii) The **schema artifacts** are bound to a concrete general schema and are used to generate transformations or schema documents.

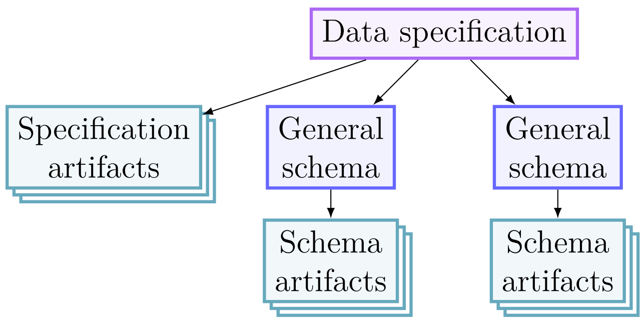


Figure 2.10: Schemas, documentation, and other generated files are artifacts. Artifacts are either schema-specific, generated for every schema, or specification-specific for a given data specification.

2.6 Inheritance

Requirement 7. The tool shall support class inheritance on a general schema level and in generated schemas.[7] That means it shall be possible to design a schema that validates data where both the base and derived classes can be used. The derived class may have additional properties.

We will start directly with an example. Suppose that the warehouse also distributes foods in addition to general goods. Food is a type of good, but it may have additional attributes for storage purposes, such as *storing temperature* or *expiration date*. Suppose that we want to design a schema for a JSON list of goods, as seen in Figure 2.11. The document is an array of objects, where each object has basic properties such as *name* and *price*. If the object represents food, we want it to have additional attributes. JSON Schema format is capable of supporting this.

```
[
  {
    "name": "Chair",
    "price": 100,
    "type": "furniture"
  },
  {
    "name": "Ice-cream",
    "price": 10,
    "type": "food",
    "expirationDate": "2022-07-21",
    "storingTemperature": "frozen"
  }
]
```

Figure 2.11: Example of JSON data we want to validate. Based on the type of good, the object may have additional properties.

[7]See Figure 8.1 for screenshots from the tool with this requirement implemented.

Without any additional information, we can only say that if the object contains only one of those additional properties, it is not valid because there is no such class that has only one of them. This allows us to validate cases when one property is missing. Nevertheless, we can add a property that specifies the type (or category) of goods and use this to validate the object. Based on the type, we are then able to use the correct class for validation, ensuring that the object has the correct properties.

This requirement impacts the application at three different levels. (i) The general schema model must have constructs representing the required problem. This is analyzed mainly in the following sections. (ii) We must somehow represent the inheritance in the user interface. By this, we mean how to show that the class has a specialization in the "bullet list" representation (see Figure 2.5). (iii) All generators shall understand them and generate a schema that corresponds to the intended result.

An advanced reader may point out that the problem can be generalized by introducing a disjunction to the schema. As we will show in the following text, this assumption is correct. However, designing schemas only with disjunction is a cumbersome and complicated task for less advanced users and does not preserve the intent of extending the class. Therefore, we still want to provide the ability to work more efficiently with the inheritance.

In ordinary cases, a more general class shares its properties with all its descendants. This can be seen in the example, where the *Ice-cream* has all properties a *Chair* has. In UML modeling and most programming languages, copying the properties is unnecessary as they are inherited. We want to achieve a similar thing in our schema representation by not polluting the page with redundant information. This, however, restricts the use of inheritance because we cannot select the order of properties in the descendant class if the properties are not shown.

To formalize the restriction, a descendant class implicitly has all properties of the parent in the same order, and those properties are before any other properties of the descendant class. This is a limitation for XML and CSV documents as JSON does not depend on the order of properties. However, order generally does not play a huge role in modeling, and if it does, in most cases, the properties are sorted from most general to most specific, hence using our proposed solution. For advanced use cases, the low-level constructs introduced later can be used.

The rule is applied through the whole chain of inheritance. If there are classes A, B, and C, where C is a descendant of B and B is a descendant of A, then C has properties of A, properties of B, and then its own properties.

In some situations, we might want to omit the parent class from a schema. Let us have a base class B and its two descendants M and N. So far, we can model a schema with all B, M, and N, where B provides properties to both M and N. B may not represent a real thing per se. It may correspond to an abstract class that serves only as a base for the other classes. In that scenario, we only want to allow M and N to be used.

Because we do not want to show inherited properties, the descendant classes must visually belong to their parent to indicate that it shares its properties.

Hence, we propose that if a class has a specialization in a schema, the specialization will be shown after the properties of the parent. If the base class should be omitted from the schema but has some properties, it will be shown visually differently.

```
class Good
  - attribute name[1..1]: string
  - attribute price[1..1]: number
  specialization Food
    - attribute expirationDate[1..1]: string
    - attribute storingTemperature[1..1]: number
    specialization Fruit and Vegetables
    ...
  specialization Drink
    ...
```

Figure 2.12: Proposition for representing an inheritance in the schema. The row with specialization classes is below the property list and does not belong to it.

So far, the internal schema model, although not formally defined, does not support constructs for the purpose of inheritance. We can build on the current proposal of the UI, but this approach is not robust enough. We want to compose the desired results from low-level constructs as it lets us use them in other situations as well.

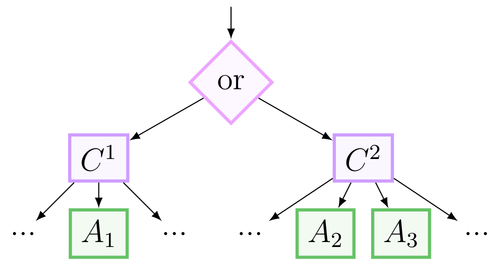
2.6.1 Disjunction in schemas

First, we introduce the concept of a disjunction. The disjunction in a schema context is a set of rules (or subschemas) where exactly one rule must be satisfied. The disjunction serves as the "OR" operator in the schema. We can use it in our inheritance problem to create an association to a disjunction of classes that have a common ancestor. It nicely solves part of the inheritance problem and can be used for other things, as well. For example, the title can be either a string or an object of language and translation pairs.

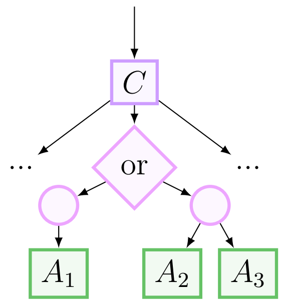
Both the JSON Schema and the XML Schema support some types of disjunction. JSON Schema has the `anyOf` keyword, which specifies that the given value must match at least one rule, effectively creating an OR. XSD has `xs:choice` element doing a similar thing.

There are two ways to implement the OR operator in our proposed hierarchical model: on a **class level** and **property level**. The former approach allows the OR operator to be placed anywhere where classes can be placed. Either in the root of the schema or in the association. The OR is then a set of classes. The latter approach is more flexible, allowing users to specify the disjunction between tuples of properties.

The former model is more common for programmers, as in some languages (such as TypeScript), it is possible to specify a type of property in this particular way as an OR of multiple types. On the other hand, the latter approach is more well known in data modeling, as XSD's `xs:choice` works exactly the same way.



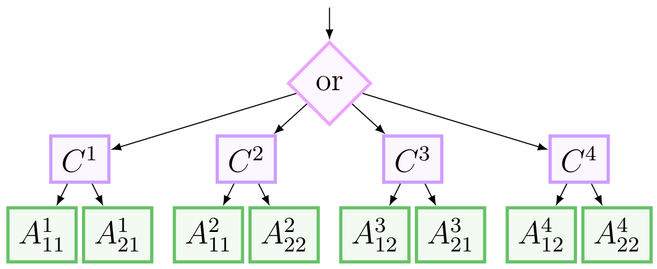
(a) Class-level OR



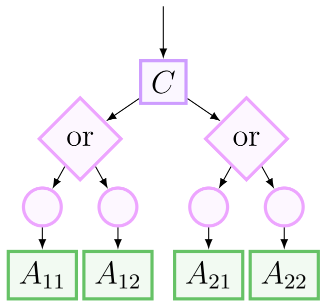
(b) Property-level OR

Figure 2.13: Comparison of two models of the semantically same subschemas of class C having either attribute A_1, or both A_2 and A_3.

The model using a property-level OR cannot use the disjunction in the root of the schema, which is an essential disadvantage as there may be use cases for those schemas. On the other hand, this model is better suited for the Cartesian product of multiple disjunctions in a single class. Suppose a schema for class C having the following attributes: the first attribute is either A_{11} or A_{12}, and the second attribute is either A_{21} or A_{22}.



(a) Class-level OR



(b) Property-level OR

Figure 2.14: Comparison of two models for the Cartesian product of disjunctions.

As seen in Figure 2.14, the model with a class-level OR tends to have wider trees for Cartesian products of disjunctions because we have to create (automatically, of course) each combination.

Moreover, the "attribute or association" construction with class-level OR is harder to achieve as we need to use the OR on the parent class with one class having the attribute and the other the association. In the case of property-level OR, this can be done locally.

2.6.2 Include in schemas

Before proceeding with the disjunctions further, we will solve the rest of the problem with inheriting properties. We already stated that we do not want to show the inherited properties in descendant classes as it is redundant information. Nevertheless, we still need a construct specifying that the properties are inherited from other classes as we only have OR with no semantic meaning.

The most straightforward way is to implement classical inheritance, as is known from programming languages, between the physical classes in the model.

However, this limits us in some schemas where the order of the attributes matters. Therefore, we will use a new construct **include**, which can "copy" all properties of the given class and insert them in the place where the include is located. Include is hence a part of class properties alongside attributes and associations. The include with the class-level OR can be fully used to implement the desired inheritance.

All classes that participate in the inheritance are internally under the same OR, as only one of those classes is used in the resulting data representation. Each class, except the base class, has an include as a first property in the property list. The include then points to the nearest parent class.

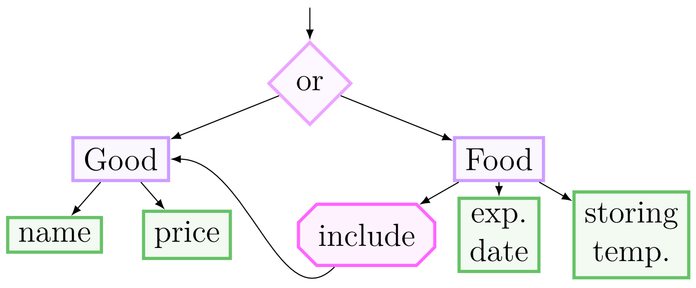


Figure 2.15: Proposed schema model that handles the inheritance problem with an include and an OR constructs.

Besides the common use, having the include construct allows us to overcome the problem of the cartesian product of disjunctions, which is shown in Figure 2.14. This is, however, not a proposed solution as we currently do not have a use case where solving this problem is important. Because the include extracts properties of the included subject, we may combine include to OR to a set of classes, which according to the defined logic, would extract properties of one of the included classes.

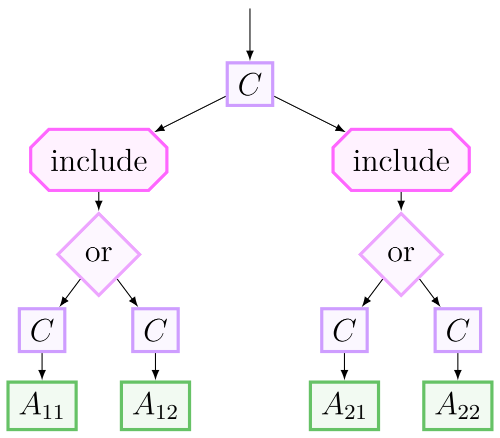


Figure 2.16: Using include-to-or construction to achieve the same thing as in Figure 2.14b. The OR selects one of the two classes and the include copies the content to the parent.

2.6.3 Type coherency

Although both OR and include constructs add complexity to the model, it is still possible to ensure basic type safety rules, which are necessary to check the model's correctness and provide relevant information for the user. For example, if a user would like to add a class to OR, the UI shall list only the classes that are suitable in that context.

To begin with include, its purpose is to take all properties of a given class and insert them in the place where include is located. Suppose that class A includes B. As A gets all the properties of B, B's type, as the **included class, can only be the ancestor or self**. This even nicely works with classes that were not created from the ontology (hence not having a type), as those classes may contain some technical properties that can be included by any class.

By doing a similar analysis with OR, as OR is linked to an association, each class inside that **OR may have a descendant or self type**. Hence, we say that the type of OR is the nearest common ancestor of all classes. This works well with root as well, as it can have an arbitrary type; hence the classes in the OR can also have any type. Nevertheless, for that scenario, we would probably like to restrict the type to a parent class anyways, as this is mostly the desired behavior.

It may seem that things work. Unfortunately, there is an exception when we combine these two constructs together. As we have already shown in Figure 2.16, using include-to-or may be beneficial but does not work with the introduced rules as include requires ancestor or self type, but OR provides descendant or self type. However, this is specific only to include construct, which can easily be fixed by having two kinds of ORs, let's say generalization OR and specialization OR. We will keep these problematics for future work as there are not enough use cases to evaluate whether this approach is beneficial or not. So far, we thus only allow specialization OR that can be placed into a root, at the end of the association, or can be referenced from another schema.

3. Future requirements

The following requirements in this chapter are analyzed because they may affect the final model that will be discussed in the next chapter. But due to its complexity, full implementation and analysis will be kept as authors' future work, and this thesis covers only the necessity not to introduce technical debt. The requirements follow the authors' intention of creating a whole ecosystem that supports advanced features of sharing and managing schemas.

3.1 Ontology modifications

Requirement 8. The approach of previous tools for creating the ontology directly in the application is not required, but there should be support for *some* modifications.

As stated in Requirement 2, the preferred way is to create a complete ontology externally and keep it up to date and valid against the requirements of all involved parties.

However, there may be scenarios where it may be beneficial to change the ontology directly. Some of them are the following:

1. The ontology is wrong and does not describe the domain correctly. - *Then, a correct way would be to fix the ontology.*
2. The ontology describes only a subset of the domain. Either only the core of the domain or the ontology is complete, but only for one domain, whether in another, something may be missing. - *If the desired ontology is strictly a superset of the domain, we can exploit the linked data features to add missing annotations in our own structured data. Then we would use the new ontology.*
3. The ontology is not granular enough. Some entities can be represented in more detail than they currently are or vice versa. - *We would need to create a copy of affected classes or use an advanced tool if it exists.*

Suppose the example with goods in the delivery company. Although the goods may be identified by EAN (barcode on items), the software team may prefer their own internal identifiers. There may be reasons for not including the identifier in the ontology, as it is too specific for only a software team, for example. This would correspond to the second category from the list above. The missing attribute then may belong to either the original class or the new extended class in the modified ontology. The third category may represent the case where, for example, we need to replace an address with a set of more specific attributes such as *street*, *number*, *city*, *country*, etc.

Although in all scenarios, the preferred way would be to create a new ontology or modify the remaining, it can be too cumbersome and time-consuming, especially if the data modeler wants to try something with an altered ontology, or if a small error needs to be fixed quickly. Therefore, the opportunity of modifying the ontology should be possible.

Allowing such changes must be done carefully as it may interfere with some mechanisms.

1. If the original ontology changes, the local overwrites may need to be changed as well; otherwise, they may become invalid. Overwritten data may be removed or moved elsewhere. The evolution mechanism (that is described in more detail in Requirement 10) hence must work with the overwrites as well.

2. Moreover, the overwritten data may change, which can lead to two scenarios. Either the user wishes to keep the local version as if nothing happened, or they may want to discard the local version as the new version fixes the issue that caused the modification in the first place.

This issue is too complex, and due to the nature of the requirement, it must be solved directly in the application. We keep the question behind this problem open and focus only on simple modifications, as this will cover most use cases.[1]

3.2 Ontology alignments

Requirement 9. As there shall be a support for data transformations between different schemas, the data transformations shall respect various ontology alignments to transform data between different ontologies. Alignments shall also be created during user modification of the ontology, between the modification and the original ontology.

Alignment, as defined in [5] is a set of relations between entities, *usually* from different ontologies. These relations specify the semantic equivalence between them and create a mapping that can be used to transform data from one representation to another.

There are already well-known RDF predicates that can cover basic alignment. For semantically identical entities, we may use `owl:equivalentClass` or `skos:exactMatch`. A more useful RDF predicate is `rdfs:subClassOf` to specify that the given class extends others by specializing its type.

The latter is already used in the previous Requirement 7. Subclasses (i) reuse attributes and associations from their parent class but also semantically denote that (ii) the subclass can also be treated as "the parent class." The second point is an example of a simple ontology alignment. In the context of data transformations, we can take the more specific class and convert it to its base type.

As an advanced example, suppose again the address property that is used for delivery purposes. The address can be represented as one multi-line string value or as its parts such as *street*, *number*, etc. The most straightforward mapping would split the string by commas and new lines and join them back, respectively.

The user can then decide whether they want to use the first or the second group of entities, and the transformation script would still be able to transform data between those two representations.

[1]From our specific use case on the Semantic government vocabulary (SGOV), most local changes consist of adding a missing cardinality or fixing labels and descriptions.

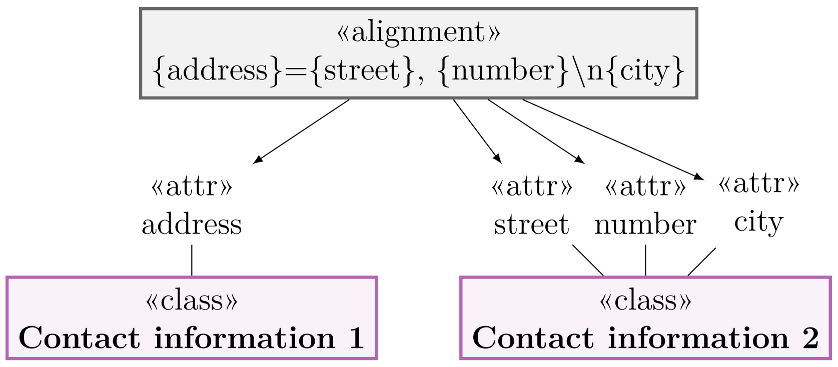


Figure 3.1: Example of mapping between different representations of address.

The primary purpose of alignments is to use them for better data transformations, as different general schemas may use different parts of the ontology. By connecting them with alignment, we may be able to transform the data between a wider variety of schemas. We shall note that using multiple syntactically different ontologies in modeling is not our use case as our primary focus is to use ontology designed primarily for modeling. Nevertheless, there are some scenarios where some alignments are useful.

In general, we can use alignments directly from various ontologies (see Requirement 2) if the ontology supports it. As we have pointed out, we already use subclassing from the supported ontologies.

In addition to explicit use, alignments are also crucial for local modifications (see Requirement 8). As we decide to introduce a new entity or modify others, it will be beneficial to keep the information that those entities are somehow related to the original ontology. Suppose the example with the address. We find that the address as a single field is not sufficient. Therefore, we split the address into individual parts and use them. The alignment together with the transformations still produces valid RDF data according to the original ontology.

This approach can even be used to create transformation scripts between old and new data if the ontology changes (see Requirement 10).

3.3 Evolution

Requirement 10. It shall be possible to perform an evolution of schemas and other artifacts from an ontology. The evolution shall be automatic, if possible, and shall also transform the data that conform to the given schemas and deduce the changes from an ontology that does not support versioning.

Although designing the schemas with the documentation may seem like a one-time job, later management of the schemas is also essential. User requirements may change, resulting in a change in the ontology and underlying schemas. The change may be as simple as adding a new property but can also be more complex, such as splitting classes, moving attributes, or changing their semantics.

As the tool's purpose is to support the whole process of designing the schemas, it shall also provide the possibility to change the schemas in the future easily. We can analyze this requirement on two levels: how to change the schemas and how the change is reflected.

Our current goal is not to create a complex model capable of any change but rather to create a simple, easy-to-maintain solution that can handle most cases. Moreover, for complex changes, it may be cleaner to recreate the schemas from scratch without the need for any evolution mechanism.

Changing the schemas The source of the change is the ontology, as we are interested only in a top-down (from the ontology to schemas) modeling. Because we want to support all kinds of ontologies, we cannot have additional requirements, such as the history of changes. Therefore, the tool needs to have a mechanism to analyze the ontology in the current state and generate a list of changes.

Having a list of changes and the previously designed schemas, we can perform the evolution. Depending on the context and the user preferences, some changes may be performed automatically. For example, suppose that *name* of the goods is changed to the *title*. This change is simple, and since we are performing the evolution, we probably want the change to be applied as is. On the other hand, some changes may be more complex, where user interaction is necessary.

In any case, the result of the evolution is a new general schema that conforms to the ontology. We can use this general schema to re-generate documentation and schemas for desired languages. In some cases, this may be sufficient, and the work ends here.

Reflecting the changes Nevertheless, some users may not be satisfied with just a new version of schemas and documentation, as it may be difficult to find out what has changed and how. To painlessly apply the changes in their systems and to understand the change, they may require:

1. **Data transformations between the old and new schemas** to easily convert the data to a new representation. This may be useful as a temporary workaround to switch to a new format without actually changing the application that uses it. Transformations, of course, can be used to convert all data to the current format if data are stored in it.

2. **A document describing what has changed** to easily understand and apply those changes. The document format can be, for example, an HTML file containing the table of renamed attributes, associations, and classes with a textual description of more advanced changes. The purpose of the document can be similar to the documentation and may link other documents and transformation scripts.

Data transformations are de facto already handled by the previous requirements. We will not modify the existing schema during the evolution but rather create a copy. Because both schemas use the same ontology (possibly with alignments), we can generate data transformations between them with RDF as the central format.

Generating documents would probably require a new type of generator that would work on two schemas at once. This, however, is too complex for the current state of the project. Therefore, we will keep this problem for later.

3.4 Interoperability

Requirement 11. The tool shall support working with general schemas that are not directly stored in it but may be located in another instance, on the web, or in Solid Pods[2].

We have already discussed data on the web principle regarding ontology (see Requirement 2), as it is preferred to have data published on the web to be easily accessible by anyone. Although this can be achieved in other ways (such as public API, regular dumps, etc.), the great benefit lies in the fact that those data are independent of the tool that created them. Those data can be easily modified and accessed by other tools (if the given tool understands its structure) and, if necessary, moved under a different database.

Similarly, we would like to achieve this with all data representing the schema's state. Specifically, we mean the structure of the general schema, configuration of all artifacts, other configurations, and helper files. Instead of having an enclosed application that stores all data internally and only provides a way for exporting and importing them, we would like to have ways to read schemas from other sources similarly as they are local and modify them as well if the user is allowed to do so.

This approach allows data modelers to create their own schemas that can be reused by anyone else on the internet. Because the schemas would be hosted by their infrastructure, there is no need for a centralized service that would need to deal with user accounts, GDPR, payments for schema hosting, integration of other tools, etc. Of course, this also means that there would be no repository with search functionality for the schemas.

In most cases, storing data externally should not be a problem, as we need to read them from somewhere anyway. If the external storage is inaccessible, the application shall still provide most of its functionality and try to obtain the data later. For example, this may mean that it would not be possible to generate some artifacts, and part of the schema in the UI would not be visible. Because we have introduced data specifications as projects, the problem would only occur when referencing a subschema from a data specification that is stored in the problematic source.

This approach may be challenging if we begin changing the schemas. In the current state of the implementation, schemas can be referenced. If the referenced schema changes (either by evolution or directly by user), the reference may become broken, and referencing schema becomes invalid. In sections 2.3.1 and 2.6.3 we already tickled type coherency and when schema becomes invalid.

This, together with the fact that schemas may be modified outside the tool, has significant implications as some checks on schemas must be performed continuously and not just during the construction of the schema. Generally, that would mean that schemas may be invalid/broken at any time, and the tool shall still be able to work with them.

[2]Solid (`https://solidproject.org/`) is a specification for storing data in decentralized places called Pods. Users may create Pods in their own servers or use services that provide that option. It is an alternative to services like Facebook or Google that stores data on their servers only.

3.5 Schema inheritance

Requirement 12. It shall be possible to extend any existing general schema by adding or modifying some of its parts. The extended schema shall remain linked to the original one and allow propagation of changes if the original schema is modified.

As an example, suppose someone designs and publishes a general schema (not the generated JSON or XML schemas, but the data specification with the general schema itself).

- The most common scenario is that we work with data that conforms to the schema as is. For example, the author of the schema publishes the data in one of the formats, and we only need to process them. For this, we only need to generate schemas from the published general schema.
- An advanced scenario is that we need to wrap the data and send them elsewhere. Hence we need to create a new schema containing the original one. In this case, the schema reference (see Requirement 5) is sufficient as we do not modify the content of the payload. This is shown in Figure 3.2b.
- This requirement addresses a scenario where the payload is somehow modified. For example, we may want to create a proxy that removes personal information from the payload if the user is not logged in. This is depicted in Figure 3.2c. Other examples are to add a timestamp directly to the payload or add additional information to some parts of the data.

```
{
  "name": "John Doe",
  "role": "customer",
  "e-mail": "jd@example.com"
}
```

(a) JSON data that conforms to the original schema. (the payload)

```
{
  "recipientPerson": {
    "name": "John Doe",
    "role": "customer",
    "e-mail": "jd@example.com"
  },
  "message": "Summer sale!"
}
```

(b) JSON data containing the unaltered payload from above.

```
{
  "name": "John Doe",
  "role": "customer",
  "e-mail": null
}
```

(c) JSON data of the payload with censored `e-mail` as it is the personal information.

Figure 3.2: Example of the second and third scenarios from Requirement 12.

Similar to reference in schemas (Requirement 5), it shall be possible to extend any schema from any data specification. Without the need for evolution

(Requirement 10), it is sufficient to simply copy the whole data specification and modify it directly. But in situations where the data depend on other data that conforms to the specification, it is better to have schemas linked to propagate the changes automatically.

As in the previous requirements, we are interested only in minor changes, as for large modifications, it may be impossible to perform evolution, and if so, there would be many possible solutions, which would effectively undermine the whole purpose of the schema extension, which is to not create additional work for the data modeler.

Below we show a sample set of operations for which, under some conditions, it should be simple to implement the evolution. The detailed analysis of the problem is left for future work.

Removal of an entity If an entity is removed from the derived schema, then any changes to that entity shall simply be ignored. Change of order of properties on the parent class can be performed without a problem simply by applying the new order without the removed property (as the entity must be connected to some class by association). Nevertheless, if the entity is later used somewhere else (for example, in another class by including it), there can be two appropriate actions. Either not include it as it was removed or include it normally as it was meant to be removed from the parent's property list only.

Addition of new property Creating new entities does not bring any issues as those entities cannot collide with those from the child schema. If the entity is added to a list of properties, it is still possible to change the order in the parent schema as the added property, for example, can keep its absolute position in the list.

Changing the options Restricting cardinalities, changing titles, and specifying names and descriptions should be possible. If the parent schema changes those values, the tool shall ask the user whether to accept the change or not.

4. Formal background

This chapter proposes modifications to the framework layers introduced by previous tools *XCase* and *eXolutio* and formally describes them. Some problems are further analyzed as the reasoning depends on the framework structure and not just user requirements.

As mentioned in previous chapters, not everything has been implemented yet due to the complexity of the problem. Nevertheless, it is crucial to properly design and plan everything in advance to minimize the technical debt.

In contrast with the approach introduced in *XCase* and *eXolutio* tools, the process of creating the domain ontology is moved from the application to the external tools. The application then only uses those ontologies if required.

To fulfill the introduced requirements, we have modified the previously introduced five-level framework in the following way:

- We have added a new top-most level **CIM** (from *Computational Independent Model*). CIM represents the remote ontology on the web according to Requirement 2. Although the level is part of the framework design, it is important to stress that it has no direct representation in the tool as it represents data on the web. Because we suppose ontologies respect LD principles, we can see them as a single graph, not multiple independent sources. To be strict, our definition of CIM does not correspond to the definition from MDA, as CIM shall be only a concept with no representation. Hence our CIM is more like an online-PIM level, but due to simplicity, we keep the naming as we proposed.

- The previous **PIM** layer is used as a copy of the CIM layer, and only the necessary entities are copied to it. This approach is compatible with the design of the previous tools, which used PIM as the source of the ontology.

This modification brings several advantages:

1. As the ontology is copied, we can use the tool seamlessly without depending on the ontology. We can generate artifacts and modify the schema. Only the operations related to directly using the ontology depend on CIM.

2. The mechanism that derives a list of changes during the evolution (see Requirement 10) may use the PIM layer as a comparison.

3. The layer still separates the ontology from the rest of the model, simplifying the design of the whole framework. For example, the other layers may refer to information in PIM.

PSM, as a second level from the five-level framework, then represents the general schema.

In previous chapters, we defined data specification as a project containing multiple schemas, configurations for generators, and metadata, such as a list of reused specifications. This would mean that individual PSMs belong to a concrete data specification. To simplify the architecture, we state that exactly one PIM is part of the specification with multiple PSMs.

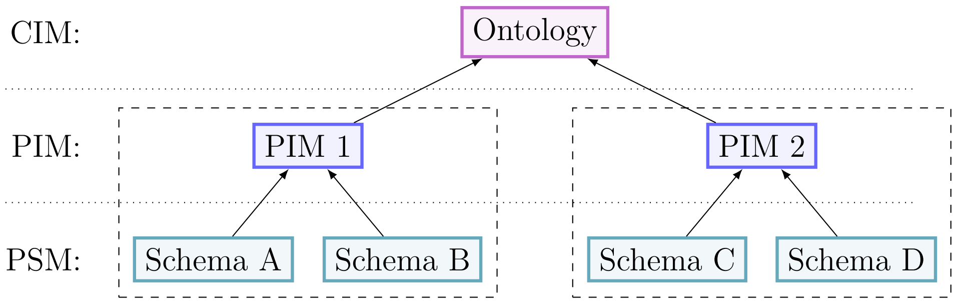


Figure 4.1: Schema of the new framework structure (without the lower levels containing artifacts). The topmost level represents the ontology; then, different PIMs serve as copies of the ontology for different data specifications that are symbolized by dashed rectangles. The third level represents the general schemas.

4.1 Conceptual levels

We will start by defining PIM, as the definition of CIM depends on it.

Definition 1 (PIM)**.** PIM is a quadruple $C = (C_\text{c}, C_\text{attr}, C_\text{assoc}, C_\text{end})$ of sets of classes, attributes, associations, and association ends, respectively (we will call them as entities), with a set of annotation functions such that:

- Attribute $a \in C_\text{attr}$ belongs to exactly one class $c \in C_\text{c}$, which is denoted by annotation function $\text{class} : C_\text{attr} \to C_\text{c}$ as $\text{class}(a) = s$.

- Association $r \in C_\text{assoc}$ has a tuple of two distinct association ends denoted by $\text{end} : C_\text{assoc} \to C_\text{end} \times C_\text{end}$. Each association end belongs to exactly one association. Each association end has one class defined by $\text{class} : C_\text{assoc} \to C_\text{c}$.

PIM entities can be decorated by other various semantic and syntactic annotations. We do not require that an annotation must be defined for every entity if not stated otherwise.

- Classes, attributes, associations, and association ends may have title and description, (for example $\text{title}(c) =$ "Tourist destination"@en) or potentially other describing properties that are not directly used in schema generation. However, the title may be used to propose the naming of entities' labels at the PSM level.

- Each class has a set of classes that extends by annotation $\text{extends} : C_\text{c} \to \mathcal{P}(C_\text{c})$. (see Requirement 7)

- Attributes and association ends have cardinalities $\text{card}_\text{min} : C_\text{attr} \cup C_\text{end} \to \mathbb{N}_0$ and $\text{card}_\text{max} : C_\text{attr} \cup C_\text{end} \to \mathbb{N} \cup \{\infty\}$, where $\text{card}_\text{min}(i) \leq \text{card}_\text{max}(i)$, where the comparison operator works the same as in the extended real number system.

- Attributes have data types $\text{datatype} : C_\text{attr} \to D$ where D is a set of data types, usually specified by an IRI.

The purpose of annotations is to bring additional information to the model that is not essential for the generation. As we stated in the previous chapter, there are various ontologies, some of which may lack the support of some construct. We have already mentioned that RDFS does not allow naming the reverse direction of an association. Some ontologies may not support cardinalities or inheritance (although most of them do). Similarly, some artifacts may not use all the information from PIM. For example, data transformations do not need a title and description to work with.

We do not provide a complete list of all annotations as the intention is to let programmers use their own if necessary, either when creating a new generator or adding support for the new format of an ontology.

The purpose of PIM entities should be clear, as we have described all essential concepts in chapter 2.

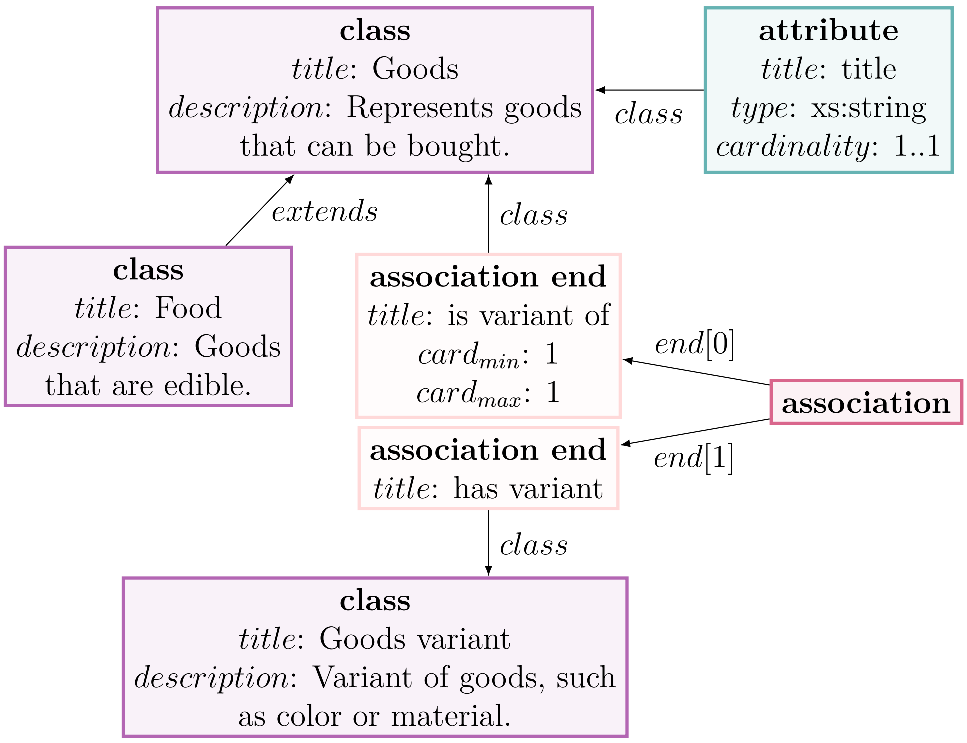


Figure 4.2: Example of the PIM model shown as a graph. Rectangles represent entities. An Italic font inside the rectangle or on the arrow represents the given annotation function with the given value.

Because during the modeling process, CIM (specifically ontologies under different formats) is being copied to PIM, it would be reasonable to define the CIM in a way that is compatible with PIM.

First, we need to define an interpretation that will be used to connect entities from PIM with those in CIM.

Definition 2 (interpretation)**.** Let us have an annotation interpretation : $E \to \mathcal{I} \cup \{\emptyset\}$ from all entities to $\mathcal{I}$, a set of CIM entities. We say that PIM entity I is interpreted if and only if annotation interpretation$(I) \neq \emptyset$.

Definition 3 (CIM)**.** CIM O is an ontology for which function **CIM adapter** A exists, such that $A(O) = C$ is a valid PIM, where every PIM class, attribute, and association has a different defined interpretation representing the CIM entity, and that interpretation is stable over time. That means if the CIM entity is changed but still represents the semantically same thing, then the interpretation of the corresponding PIM entity shall stay the same.

The definition tells us that the CIM can be viewed as a PIM with an interpretation as a pointer to the original thing in the ontology. In practice, it is the IRI of the entity in the ontology.

If the CIM changes, entities shall keep their original IRIs to stress that the given entity is semantically still the same. Only the representation may have changed. This will help us, for example, to detect changes and properly propagate them in the model.

For simplification, in the rest of the thesis, we may omit that CIM *needs to be transformed* to PIM and suppose that it is already in PIM-like format.

The definition does not require association ends to be interpreted as some ontologies consider the whole association as a single entity. Nevertheless, each association end belongs to its association which is interpreted. Hence the link to CIM exists.

4.2 Structural level

Ontology is represented on conceptual levels on CIM and PIMs. The constructed schemas then belong to the structural level. During the analysis of Requirement 1 we have already decided on the hierarchical structure of the general schema as our target schemas also have a hierarchical structure.

We will introduce PSM (platform-specific) level to represent the schemas. PSM level is highly inspired by the PSM level from the previous tools, but we must take into account different serialization formats PSM level is generated into.

Definition 4 (PSM)**.** PSM is a tuple $S = (S_\mathrm{r}, S_\mathrm{c}, S_\mathrm{ref}, S_\mathrm{or}, S_\mathrm{attr}, S_\mathrm{end}, S_\mathrm{incl})$ with a set of annotation functions such that:
(let $C := S_\mathrm{c} \cup S_\mathrm{ref} \cup S_\mathrm{or}$ be a set of **objects** and $P := S_\mathrm{attr} \cup S_\mathrm{end} \cup S_\mathrm{incl}$ set of **properties**)

- $S_\mathrm{r} \neq \emptyset$ is a set of roots of the schema, each having annotation $\mathrm{root} : S_\mathrm{r} \to C$ specifying the root object.
- S_c is a set of classes with annotation function $\mathrm{parts} : S_\mathrm{c} \to P^n$ that returns a tuple of class properties. Each property belongs to exactly one class.
- S_ref is set of all references to other PSMs with annotation function $\mathrm{ref} : S_\mathrm{ref} \to \mathcal{S} \times \mathcal{S}_\mathrm{r}$ that returns the referenced PSM and one of its roots.
- S_or is a set of ORs with annotation function $\mathrm{choices} : S_\mathrm{or} \to \mathcal{P}(C)$ that returns the set of all possible choices of the OR.
- S_attr is set of attributes with annotation function $\mathrm{technicalLabel} : S_\mathrm{attr} \to \mathrm{string}$ that returns the label of the attribute.

- S_{end} is a set of association ends with two annotation functions (i) technicalLabel : $S_{\text{end}} \rightarrow$ string that returns the label of the association and (ii) end : $S_{\text{assoc}} \rightarrow C$ that returns the associated object.
- S_{incl} is a set of all includes with annotation function includes : $S_{\text{incl}} \rightarrow C$ that returns the included class.

The definition puts together the findings from previous chapters, where we analyzed the structure and individual concepts of the general schema. The definition re-introduces well-known terms from PIM, such as class, attribute, and association end. Association itself has no counterpart in PSM as we are only interested in one specific direction. References are a necessary concept for the implementation part as their meaning is to reference outside of the PSM, whether referring to entities inside PSM is implicit. As decided, class-level OR is used to support disjunction, hence belonging to object types that can be associated and placed as schema roots.

For advanced cases, we allow multiple roots for a single schema, such as creating a database model of multiple tables. For most schemas, however, only one root is allowed. Hence, the general schema with root R is PSM S with a single root $S_{\text{c}} = \{R\}$ with all other entities being part of the chain originating from the class.

Although it may seem that the PSM is a forest (tree for every root), the PSM is neither DAG nor a connected graph. We have already discussed the include construct (see Figure 2.15) as it takes properties from another existing class, hence creating a diamond shape in the graph. We also did not restrict that two different association ends may point to the same object. Nevertheless, we also allow oriented cycles. For example, a class may have an association end referencing the class itself. This allows us to design schemas for data structures containing, for example, serialized trees, as a tree can be arbitrarily deep. All properties, on the other hand, such as attributes, association ends, and includes belong to a given class, and hence may not be referenced multiple times.

To keep the relation between entities from PSM and PIM, we will introduce the same concept of interpretation for PSM with a slightly different meaning. On PIM, interpretation means that the entities are de facto the same. On PSM, however, we need to say that the given PSM entity is only semantically the same as the corresponding PIM entity.

Definition 5 (interpretation on PSM)**.** Let us have an annotation interpretation defined as follows on the set of PSM classes $S_{\text{c}} \rightarrow C_{\text{c}} \cup \{\emptyset\}$, attributes $S_{\text{attr}} \rightarrow C_{\text{attr}} \cup \{\emptyset\}$ and association ends $S_{\text{end}} \rightarrow C_{\text{end}} \cup \{\emptyset\}$.

Interpreted PSM entities are linked to their PIM counterpart, which semantically means that they were constructed from it. ORs, includes, and references cannot be interpreted as they do not represent concepts from the ontology.

Both definitions of interpretation annotation imply the existence of non-interpreted classes, attributes, and associations. We will discuss the meaning of the non-interpreted PIM entities later in subsection 4.3.2, but non-interpreted PSM entities might be used to introduce additional properties to the schema.

For example, suppose that our data are wrapped in another class with properties *payload* and *status*. *Status* informs if the request succeeded and the *payload* contains the required data. Then, the wrapper class, the *status* attribute, and the *payload* association would be non-interpreted.

4.2.1 Format-specific PSM constructs

It is essential to mention that both PIM and PSM may be extended in the future by adding new types of entities. This was already indicated in Requirement 9 that the alignment construct might be added to PIM.

Whereas this is not causing issues in the PIM, as an additional construct may be simply ignored if not understood by the rest of the application, we must proceed with caution if we want to extend PSM. PSM schema must be understood completely to correctly generate artifacts from it. Hence even a small change may impact many other parts of the application and also possible third-party plugins, which are expected to read PSM.

This is an issue only if we intend to generate schemas in various formats from the PSM. If the use case is to use PSM to generate a single format, such as XML schema, then there is no problem with using XML-specific constructs that may break the generation of JSON and CSV schemas. Nevertheless, it is advised that any format-specific option shall be used as an annotation, if possible, to not break the generators.

As an example, XML schema has `<xs:sequence>` and `<xs:all>` model groups that specify whether the elements are ordered or not. Although on the XSD level, those are different things, on the PSM level, we may introduce annotation $\text{ordered} : S_C \to \{\text{false}, \text{true}\}$ to achieve the same effect.[1]

On the other hand, one may want to add support for comments. Although comments are usually intended to be bound to a specific element, hence annotation might be enough; it is possible to introduce them as standalone entities. This would require reimplementing all generators as they need to understand the concept of this new entity.

Similar to PIM, PSM entities, as well as the PSM itself, may have additional annotations. Besides that, generators may exploit interpretation to PIM to obtain additional info about entities. This usually means, for example, that interpreted PSM entities do not need to define title, description, or cardinality, as those can be extracted from PIM. We still must allow those annotations on PSM for non-interpreted entities or for overrides.

Regarding the definitions It may seem that, in general, the definitions are too permissive or incomplete. This is intentional, as any introduced restriction may limit some functionalities in the future. Hence, we prefer this robust model and move the burden to generators to decide whether the schema is invalid in the current context or whether generating an artifact with a warning message is still possible.

[1]This is not entirely correct as XSD may specify that only some elements are ordered, whereas others may have random order. This is an example of a feature that is currently too complex to be handled by our general schema model.

In fact, from the perspective of incremental development of the tool and extendibility of the model, this is a preferred behavior. Suppose, for example, that some generators may not understand the concept of OR, as it is not trivial to handle. In that case, the generator may simply ignore it and generate at least the rest of the schema/artifact. Users then get an incomplete result (with a warning) which they may fix by hand. A similar rule is applied implicitly to annotations, as generators use only the known, keeping the other ignored.

4.3 Changes on the framework levels

4.3.1 Changes in CIM

So far, we have introduced PIM and CIM layers and only tackled how the framework would work. Before we move further, we will analyze how Requirement 10 on evolution and 8 on ontology modifications would impact the framework.

Because the CIM is used only for building the PIM, the tool does not need to know that the ontology has changed as it works mostly only with PIM. However, to further expand the schema, the tool must fetch other parts of the ontology, which may collide with the PIM.

Having PIM strictly as a copy of CIM may help in this process as we can compare the two levels and, based on the difference, *somehow* generate a sequence of operations that modifies the PIM. The modifications may be as simple as *changing a class title* or *changing an association cardinality* to the complex ones, such as *joining two attributes into one* or *moving an attribute to another class.* It is essential to have the changes that complex as they carry the information on how the schemas and their data should be modified, not just the final state. We will describe the operations later.

To formally describe the difference, we will introduce the concept of consistency.

Definition 6 (consistency)**.** We say that the annotation of interpreted PIM entity is consistent with CIM if the corresponding CIM entity exists and the value is equal to the value in CIM or, in the context of the annotation, is a superset[2] of the value in CIM. We say that interpreted PIM entity is consistent with CIM if the CIM entity exists and all annotations are consistent with CIM. Finally, PIM is consistent with CIM if all interpreted entities are consistent.

Based on the use case, most of the changes in the CIM that are worth propagating are simple and well-isolated. This may ease the process of inferring differences between PIM and CIM by finding those isolated sets of entities and then creating a sequence of changes based on a predefined set of rules.

As mentioned in the analysis, for complex changes, it would be easier to generate a delete-and-create set of operations instead of trying to figure out what has changed. This would remove affected entities entirely from created schemas after propagating, and a user would add new entities back to desired places in the schema structure.

[2]Formally, each annotation shall define its own rules, whether is consistent or not. Consider, for example, extends. As we want PIM to be a subset, we allow extends to be a subset of all classes the given class extends.

We will leave the problematics of making PIM consistent for the authors' further work. So far, we will suppose that the CIM is constant and cannot be changed.

4.3.2 User modifications

Direct modification of PIM (see Requirement 8) may break the previous approach because the mechanism that tells us what has changed would also try to revert all the changes made by the user.

To be more specific, we are interested only in those entities that have interpretation - entities that are linked to CIM. All other non-interpreted entities are so far ignored.

Formally, it may seem that changing an interpreted entity (and still keeping it interpreted) should not be allowed as the changed entity itself does not represent the ontology anymore. As this may be true, there are still some cases when the change is necessary, especially when the CIM does not give us all the information we need (such as missing cardinality or missing description) or there is an obvious error that needs to be fixed.

Nevertheless, we expect only minimal changes to be made by the user because of the abovementioned reasons. Due to the same reasons, those user modifications shall be checked every time the PIM is being made consistent because the change in the ontology may fix the same problem as the user modification, hence making the modification resolved and irrelevant.

Because of that, we **allow editing of the PIM directly** as this is the simplest option that will satisfy the requirements under the expected use case.

Create and update operations are simple and are summarized below. To make modifications complete, we also need to delete the entities. As we have defined PIM as a subset of CIM, simply deleting the entity would not be enough, as we would not know whether the entity is deleted or just not discovered. Therefore we introduce a new annotation that marks the entity as deleted. Deletion is a purely cosmetic feature, as, without it, the sufficient approach would be to ignore the entity. It only forbids users to use it in the lower levels.

Definition 7 (deleted)**.** PIM annotation deleted : $I \rightarrow \{\text{false}, \text{true}\}$, where I is a set of interpreted classes, associations, and attributes, denotes that the interpreted entity is deleted and must not be used in the lower levels.

The introduced approach provides us with the following options for modifying the PIM:

- We can **create** new entities by simply adding them to PIM without interpretation.
- Existing entities can be **edited** directly while keeping them interpreted. This, however, will collide with the evolution mechanism and must be explicitly excluded from it every time the evolution is performed.
- To **remove** an interpreted entity, we must mark it as deleted. Non-interpreted entities can be removed directly.

We have already provided reasons why modifying PIM is a relatively good idea, such as when the ontology contains minor errors that need to be fixed immediately. PIM also allows the creation of new entities, which is similar to PSM; hence it may not be clear which level to use. In general, using PIM means that multiple schemas may interpret the created entity, and therefore, it shall be possible to generate data transformations between them. Hence, entities that **seem to be part of the domain ontology shall be created on PIM**. **Entities on PSM have more structural meaning.** For example, if the schema requires some entities wrapped in another class. This is a purely structural requirement that should not have a representation in the ontology, hence shall be created only in PSM.

4.3.3 Ontology alignments

To complete the walkthrough through requirements, we must analyze how Requirement 9 on ontology alignments will impact the framework in the future.

In general, because the alignments may be arbitrarily complex, we will use a new type of construct to represent those alignments. For example, the alignment from Figure 3.1 that maps addresses between different representations would be a single PIM entity that contains that information.

Some readers may object that the annotation *extends* used for the inheritance of classes is not consistent with this approach. In the previous chapter, we said that inheritance might be considered a form of alignment. Nevertheless, unlike the other alignments, inheritance is an important concept that is used in many places in the tool. Therefore we will keep this as an annotation for now and consider reimplementation of this concept in the future when implementing the support for other alignments.

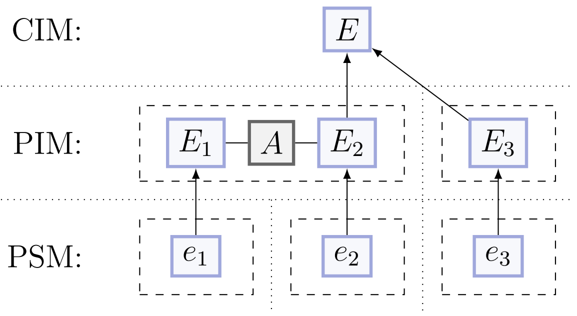


Figure 4.3: Example of three different schemas having entities e_1, e_2, and e_3, respectively. The first two schemas use the same PIM, while the third uses another. Because E_2 and E_3 interpret the same entity E (hence are semantically identical), it is possible to map e_2 to e_3 as those entities interpret the former ones. Although E_1 does not interpret E, there is an alignment A, which also allows to map e_1 to e_2 and hence to e_3.

4.3.4 Evolution

Requirement 10 on the evolution and Requirement 2 on the ontology changes brought the necessity to synchronize the layers of the framework as changes on the upper level may impact the lower levels. This means that we cannot change data on a given level simply by replacing them with new data, as it would be hard to perform the appropriate change on the other levels.

To solve this problem, we allow modifying the levels (PIM and PSM) only through **operations**. Operations are pre-defined functions that modify data on the given level and are simple enough that they can be translated to the operation on the level below.

As an example, suppose that an attribute is removed from an ontology. First, the tool compares the CIM with PIM and generates a delete operation on the corresponding PIM attribute. Depending on the context and user preferences, a user may be notified whether the tool may proceed. Then, the tool executes the operation on PIM (making PIM consistent) and transforms the operation into a set of delete operations on PSM that removes the attributes there. (There might be more than one attribute, hence multiple operations.)

We will omit a complete list of operations in this thesis as they depend on the requirements that are not fully addressed yet. However, it must be possible to transform the operation into a set of operations on the lower level.

We do not enforce that the set of operations must be minimal, and no operation cannot be composed of others. Nevertheless, each operation must be translatable to the operations below.

Having more complex operations may be beneficial, as they preserve the original intent rather than the performed steps. More complex operations may reduce the computational difficulty as some actions from the user may lead to hundreds of operations on the model, according to research on previous tools [15].

For example, *wrap PSM object with OR* may be a valid operation used for purposes of inheritance (see Requirement 7). An alternative consisting of more atomic operations may be as follows: (1) create a standalone OR (2) connect the target object to OR (3) set OR as the target object. The alternative consists of 3 times more operations and is also harder to evaluate as the *connect* operation needs to check whether the reconnection is possible due to type coherency. On the other hand, wrapping an object with OR has no precondition rules and can always be performed. Some other operations may be even more complex in their base form. As mentioned, more complex operations preserve the context, as it is clear that the intention is to wrap the object, not move it.

In Requirement 11 on schemas on the web, we have considered data specifications as an atomic unit that is always stored at once. This approach solves the problem with the execution of evolution, as the layers of the framework depend only on those in the same data specification. Hence, we can directly update them at once.

Referencing other schemas is also without problem as the only thing that may change is the type of referenced class.

4.4 Inheriting schemas

Nevertheless, there might be use cases described in Requirement 12 where reusing layers from other data specifications would be useful. Consider a data specification that models a general schema for any format. This specification is published on the web. Someone else would like to use that schema to modify it slightly. For example, to add another attribute to the given class. Now, if the author of the original schema modifies anything, we would like to modify the derived schema as well. This, however, is not possible as the tool may not be aware of the existence of the derived schema nor may not have write access to it. (Because the schemas may be stored somewhere else, for example.)

For this particular scenario, we would need to keep a list of executed operations on given schemas. The reused schema would then remember the last executed operation of the parent, which should be sufficient to recreate the steps that were performed on the parent schema in the derived schema. We have already proposed in Requirement 12 how the evolution can work for some simple tasks.

There is one prominent solution that would nicely work with the already introduced concept of annotation.

We can copy the whole PSM, where each entity gets a new IRI, with all the references between them adequately updated and an interpretation annotation set to the original entity (not PIM). This may work, but we also need our own PIM as we cannot update theirs. But having two PIMs may cause problems, as we stated that the interpretation on the PIM level must be unique; hence only one entity can represent a given thing in CIM. A possible fix may be introducing a new annotation **inheritsFrom** on both PIM and PSM, which would point to the original entity. Both PIM and PSM then would be copied with new IRIs, and the interpretation would work as right now; the copied PSM would interpret the copied PIM. This would preserve backward compatibility (as by ignoring the introduced annotation, this is ordinary PIM and PSM) and enables us to perform evolution through the inheritsFrom annotation.

The key finding here is that supporting this requirement should not require breaking changes to the framework besides storing the list of executed operations, which is useful for other purposes, such as ability to revert changes or analyze them later.

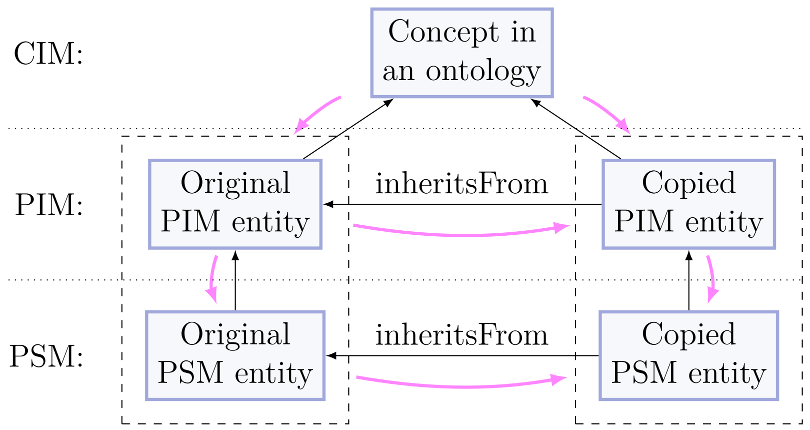


Figure 4.4: Proposition on how to handle schema inheritance. PSM with PIM is copied, and the links are preserved through inheritsFrom annotation. Possible evolution paths are denoted by pink arrows.

5. Implementation

This chapter aims to provide implementation details of the fundamental concepts introduced during the analysis in chapter 2 and formalized in chapter 4, hence closing the reasoning process. Its purpose is not to replace complete technical documentation, which can be found in a project repository (see the end of this thesis for more information).

The work intends to build a solid foundation for an ecosystem of tools with the core framework for schema modeling. The key elements that shall be followed to achieve this goal are:

1. All model data and configuration shall be stored in RDF. - *There is already an ecosystem of tools that can work with RDF. It is easily shareable and linkable.*
2. The core framework shall work on its own. - *It shall be possible to integrate into other applications. The tool is only a user-friendly interface to execute the framework.*
3. Generators shall work as plugins for the core framework. - *The idea is that anyone can design generators for their specific purpose.*
4. The model shall be robust and extensible.

Due to the current use case and state of development, our primary focus is to create a tool where is easy to design schemas and generate the required artifacts. Hence the goals above are not met yet, but some design decisions were made to fulfill them later easily.

Figure 5.1 depicts the most zoomed-out view of the tool's architecture, where the tool is shown in the context of other applications. We have noted several times that we focus only on schema modeling, whereas conceptual modeling is done elsewhere. We also plan that multiple instances of the tool may exist, each using its own database, yet still be able to provide all the functionality, such as schema reference and evolution.

Container structure Currently, the tool consists of a *backend*, *manager*, *editor*, and a *CLI service*.

The backend is a Node.js server that provides functionality for some generators, such as a documentation generator that requires a Python module to build the resulting HTML file. The backend also serves as the storage for modeled schemas. (see the bottommost layer of Figure 5.2) We also plan to use other types of storage, such as personal Solid Pods, triplestores in general, or read-only documents stored on the internet.

The manager is a React application that can manage schemas stored in the backend and execute generators. It then opens an editor under a given configuration which is another React application purely designed for schema editing. A CLI service is a command line tool used for semi-automated testing of generated schemas and transformations.

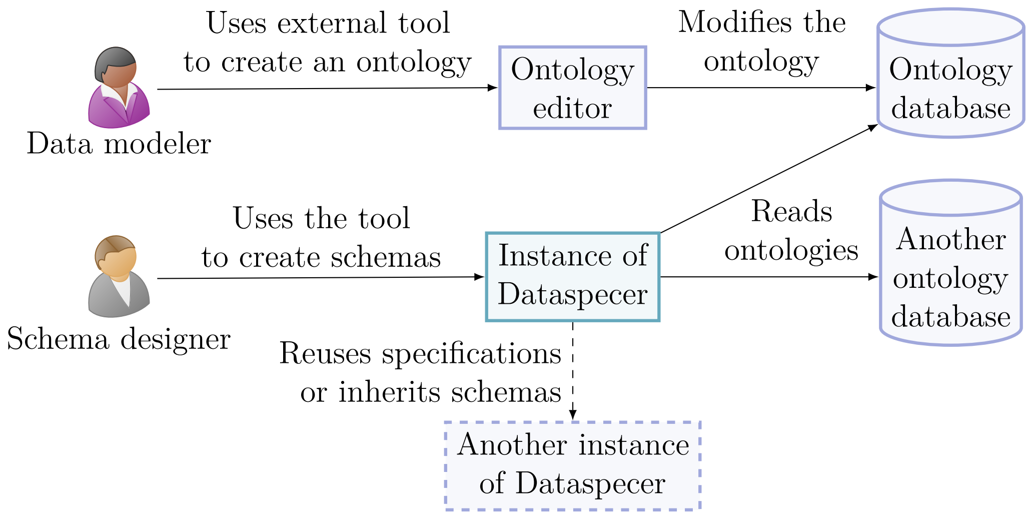

Uses external tool to create an ontology
Data modeler
Ontology editor
Modifies the ontology
Ontology database
Uses the tool to create schemas
Schema designer
Instance of Dataspecer
Reads ontologies
Another ontology database
Reuses specifications or inherits schemas
Another instance of Dataspecer

Figure 5.1: Context of the tool with other systems. Dataspecer only reads the ontology, which is modeled in other tools. The dashed line shows the intent to make the tools interoperable across the Internet as the tools shall be able to reuse and inherit specifications from each other.

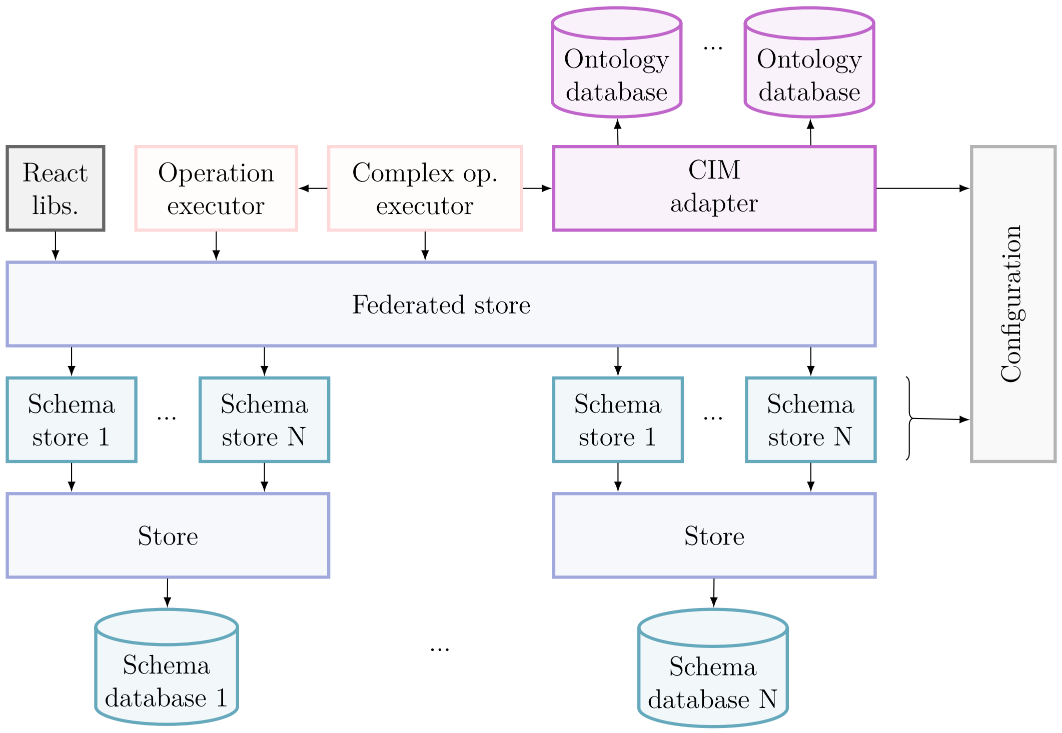


Figure 5.2: Schematic structure of the core framework and the tool. Below, there are various schema databases. Currently, there is only Dataspecer's backend, but others are planned, such as Solid Pods or Triplestores. Each database is read by store, which formally consists of schema stores containing exactly one PIM or PSM schema with all entities. All stores are merged for easier manipulation, such as executing operations. We have also implemented React library to integrate the store into React ecosystem. Based on the data specification, the configuration selects the stores that should be loaded.

Framework structure The operation execution with schema generation is bundled into the tools, as it is just a TypeScript library. Hence the tools do all the work, and the backend serves as a simple database. This is a sufficient solution for now (and necessary for most backends, such as triplestore or Solid Pods), but the structure of operations and the whole framework allows for executing the operations remotely on the server as well. This may be especially useful for large schemas during the evolution process when the difficult part can be performed safely by the server and the client only updates the local copy.

5.1 Model representation

As model data are represented by entities that shall be serialized in RDF, we introduce a **schema store** as an abstraction layer. The schema store can read and write **resources**, where the resource is a document/object that contains arbitrary data. In the context of PIM and PSM, the entity is a resource. Resources are identified by their IRI, which is an IRI of the corresponding RDF resource.

Entities can be read from the schema store, but writing is limited to **operations**. Operations that were executed are saved in the schema store to provide a history of the model at any given point in time. The schema store must contain exactly one **schema resource**. A schema resource is a resource that identifies all other entities in the schema store, formally creating a set of resources.

Schema stores are managed in **stores**. The store is an interface for reading resources by their IRI and executing operations on a given schema resource. The store is asynchronous and represents a database of resources. For example, a store may be an interface on the SPARQL endpoint, a file system, a read-only dump on the internet, or just data in local memory.

The current implementation of the tool uses stores that are synchronized with the server through a simple GET-PUT API.[1] Stores on the server are saved into individual files in the filesystem. Each store contains only one schema store for better granularity, as the file must be read and written atomically.

The store also shall generate new IRIs that can be later assigned to entities. IRIs needs to be generated in advance to be part of the operation, so we can later identify which entities were created. Depending on the store, the IRIs may have different structures.

The current implementation only supports simple access by IRI. This will be changed in the future for more advanced query operations, such as reverse lookup for the entity.

Stores' interface allows the creation of **federated stores**, hence allowing to have only a single interface for reading and writing any entity. This simplifies the application's design, as we may have a complex system of shared and reused data specifications, some of them possibly read-only, from different sources, still available under one interface.

In previous chapters, we considered PIM as the general layer (PIM layer) and the set of PIM entities we have formally defined (PIM schema). The PIM schema is represented by one specific schema store. Similarly, one schema store

[1]This de facto implicitly supports Solid Pods as a type of store.

also represents the PSM schema (possibly having multiple roots). To simplify the design, the schema resource that is necessary for every schema store will also define the PIM or PSM schema. To make the previous sentence clear, suppose PSM schema S. The schema contains entities such as classes, ORs, etc. Those entities are represented by resources. But the schema itself, which contains, for example, a set of roots, also needs to be represented by a resource. And the resource will be the schema resource.

This all means that if a user creates a data specification with two schemas, three schema stores are created: one for the PIM schema and two for the PSM schemas. Hence three stores represented by three files are made as well.

As we have noted, operations can be executed remotely. The current implementation of the store contains logic for local operation execution; hence all operations are evaluated on the client, and only the final result is sent to the server. It is, however, possible to send the operations to the server and fetch the changes. This approach is useful for better synchronization of different clients as two clients may execute operations at the same time, possibly causing a collision.

Unfortunately, we cannot go any further by executing complex operations on the server, as different schemas may be stored in different stores. A possible solution would be to introduce a proxy server, that would perform these complex operations on the given stores remotely.

Although it may seem that the whole specification may go to an invalid state after only one store was modified (for example, when an application crashes), this is not an issue. The individual schema stores are well separated and formally connected only through evolution. Hence even if only the PIM store is updated, the individual PSMs became only inconsistent, not invalid literally.

Data specification Data specifications are also identified by IRIs, but they are not resources in the same sense as PIM or PSM entities. We do not require write-through-operations access and are also read in different situations. Current implementation stores them in SQL database for fast access (which is against the first rule that everything shall be stored in RDF).

A data specification contains, namely, (i) a set of reused data specifications' IRIs, (ii) a PIM schema's IRI, (iii) a list of PSM schemas' IRIs, and (iv) a set of stores' IRIs where the appropriate schemas can be found and (v) an artifact configuration.

5.2 Layers for simplifying the model

Reading the model may be too complex, as the entities may not exist, reading them requires asynchronous access, and to obtain the value of all annotations, we usually need to go to the PIM layer. Letting individual generators access the model is, of course, necessary, but for most generators, this would mean implementing many helper functions for easier access. To reduce the burden on generators, we introduce conceptual and structural models whose purpose is to provide a more user-friendly interface for reading the model.

Conceptual model The conceptual model is relatively simple as it only simplifies access to PIM, which is simple by itself. Its structure is similar to PIM, but attributes and associations are referenced directly from the class as properties. The whole model is constructed in advance; therefore, we can check whether it is correct and not need to do it during the querying. But mainly, we can access everything synchronously.

Structural model Structural model employs a different approach to schema structure than PSM. As PSM is highly inspired by schemas, it also has a more schema-like structure. For example, include is valid class property, but from an object-oriented view, it has more semantic meaning. The structural model we use tends to be more object-oriented.

Classes also have properties, but the properties represent only attributes and associations. Include is translated to class inheritance, as it works almost precisely the same as inheriting properties from a parent class. The only difference is that the position of the include cannot be preserved. This, however, for most generators, is not an issue.

To simplify the work with disjunction, we exploit the fact that OR on one element is the same as the element without OR. Hence, all associations have an array of referred classes as individual OR choices. This won't introduce new objects that need to be specially handled and allows domain-specific generators to ignore the concept at all if it is not required. (although this is a bad design in general)

It may seem that the object-oriented interface goes against our design. Our intent is, however, to have a robust model that can handle a wide range of use-cases and rather create a layer for easier access.

Transformations Conceptual and structural models provide various transformations[2] that modify the model or obtain additional data for a given generator.

- It is possible to flatten the structural model by copying properties from parent to child classes. This transformation may simplify the work for developers of generators as they do not need to handle inheritance anymore (hence the include). Of course, this would cause the artifact to grow as the entities are not reused but may be helpful in an early stage of development as simplification.

- As some information lies in PIM, such as naming, description, default cardinality, or datatype, there is a transformation that fetches this information to the structural model.

- Associations marked as dematerialized (dematerialization is an annotation on the association that specifies that all properties from the associated class shall be put into a parent class) may be "unpacked" to the parent class instead of the association itself.

[2]Do not confuse with data transformations from Requirement 4. In fact, individual structural models can be considered as levels in MDD architecture. Transformations then correspond to MDD transformations that convert one layer into another.

Format-aware structural models As there is usually a whole ecosystem of generators that work with a given technology, such as XML, it is crucial to let the designers extend or even alter the model's interface on their own, making it format-aware. We already use this technique for XML to add namespace information to the entities. In general, the transformation may change the interface of the structural model to suit the generator's needs better. The transformation may, for example, add a new type of class property that was extracted from PSM.

5.3 Artifacts generation

We have utilized the following interface to properly generate artifacts that can reference each other, which is a crucial feature for the documentation and to keep the artifacts highly configurable. First, its configuration must be provided. The configuration consists of:

1. IRI of the given artifact.
2. IRI of the generator to be used to create a given artifact.
3. Output path, as the generated artifact has to be stored somewhere. (a directory can be provided for generators that create more than one file)
4. Public URL, as we expect, that the artifacts will be uploaded on the Internet. In most cases, the public URL shall be the same as the output path, which creates relative paths between the artifacts.
5. Generators that reference other artifacts require their IRIs. (this is necessary as, in general, we can create multiple similar artifacts, possible with a different configuration; hence the generator needs to know which to include)
6. For schema artifacts, the IRI of the PSM schema is required.

To construct the array of configurations, ArtifactConfigurator is used. The configurator itself is configured from data specification. The standard options are whether to generate all schemas, how the file structure should look or the configuration for individual generators, such as whether CSV schema shall use multiple tables for the given schema.

5.4 CIM adapter

Currently, we support only one CIM adapter (with others planned) for Semantic Government Vocabulary (SGOV) [12], as we have no use cases that would require others. In general, which adapters (multiple can be used) are used for a given schema are configured in data specification. Although this may not be necessary, and all adapters can be used all the time, we still want to limit this as the user may accidentally select a concept from a wrong ontology, and all the ontologies would have to be queried.

To follow the definition of CIM, the CIM adapter returns resources as a read-only temporary store containing PIM entities. Those PIM entities are then inserted into the actual PIM through the operations. For example, when a user searches ontology for a root class, the adapter returns a list of PIM classes from the search query. The user then picks one PIM class which is passed to the complex operation that creates atomic operations on both PIM and PSM levels and properly adds the class to the schema.

Besides the search operation, the CIM adapter provides functions for obtaining the class surroundings, complete hierarchy, or CIM adapter-specific properties. The last functionality provides data that are not stored in PIM but may be handy to display additional information about resources. For example, a URL to the resource on the Internet (as not always the IRI gives access to the resource info with human-friendly results).

Our SGOV adapter also returns tags for the resources. These tags are then shown in the UI to help the user better understand which resources to choose, as sometimes, there can be multiple resources with the same name. Nevertheless, we currently do not store the tags in the PIM to be accessible by generators. In general, this[3] would need to be analyzed concerning the evolution and local modifications and what would it mean if they change.

All resources returned from the CIM adapter are **consistent** with CIM, not identical. This means that the returned resource, for example, may not contain all extended classes (in the case of searching for the resource, this is unnecessary).

5.5 React libraries

We have also implemented React libraries for easier access to resources from the React framework. The individual resources are stored in the component state, and because the federated store implements an observer pattern, the component may listen for the changes in the store (for example from the executed operations) and appropriately update its state. All this functionality is provided from a React hook

```
const {resource, isLoading} = useResource(iri);
```

that returns the given resource and info, whether the resource is being loaded. If the resource updates, it provides the last known value until new is loaded not to cause visual flashing when a new resource is not available immediately.

The library also provides a hook for accessing any amount of resources and memoizing the result until any of the queried resources changes. This is used for advanced features, such as determining whether the OR can be replaced with a more user-friendly representation of a class having specializations (see Figure 8.1).

```
const {result, isLoading} = useResourcesInMemo(
  async getResource => {
    // Use getResource arbitrarily in the function
    // const resource = await getResource(iri);
  }, [dependencies]);
```

[3]We mean any additional properties that have no impact on the data conforming to generated schemas.

6. Related work

6.1 Enterprise Architect

Sparx Systems Enterprise Architect is a tool for visual modeling and design based on OMG's UML. It supports software engineers during the whole development process by providing constructs for modeling business logic, diagrams, the architecture of software, use case diagrams, and others. It has a proprietary license, and the product is paid.

EA supports the basics of MDA, which was introduced in chapter 1. Users can create PIM directly in the tool by modeling the relations between the classes (creating a UML class diagram), and then the tool transforms PIM into PSM representation. Transformation is, however, only elementary, and it seems that it is not possible to choose which part of PIM shall be transformed into PSM. It also does not provide the variety of options we are planning. PSM can represent XML only by default, but there is support for plugins that can extend this functionality. EA focuses more on the programming languages as, besides the XML, PSM may describe C#, C++, Java, PHP, and NUnit elements.

Although it should be possible to implement most of the desired functionality directly into EA, the proprietary license and the fact that the tool is too general for our use case makes it not unusable.

6.2 OSLO

OSLO[1] - Open Standards for Linked Organisations is an initiative that originated in Flanders, the Dutch-speaking northern portion of Belgium, to promote the use of technical standards for the data exchange between various organizations, governments, and local governments. The goal of OSLO is to maintain and create the standards through the open process (hence everyone can intervene), keep the rules respected, provide a publication platform and support the adoption of the standards. So far, OSLO contains over 18 different domains consisting of definitions from 107 organizations.

The initiative developed a toolchain to ease the standards' publication process. The standards are modeled in *Enterprise Architect UML* software and then converted into an RDF representation with their tool[2]. The following tool then generates artifacts (resulting files) that are automatically published on one centralized server `https://data.vlaanderen.be/ns`. The artifacts contain HTML documentation, RDF vocabulary, SHACL[3] templates for validation of RDF, and a JSON-LD context for developers that prefer JSON formats. Their toolchain uses the GitHub platform for storing their documents and triggering the rest of the toolchain. They also provide tools for organizations to validate their data against the standards to check that data are machine-readable without errors.

[1]`https://joinup.ec.europa.eu/collection/oslo-open-standards-linked-organisations-0/about`

[2]`https://github.com/Informatievlaanderen/OSLO-EA-to-RDF`

[3]`https://www.w3.org/TR/shacl/`

OSLO also focuses on the interoperability of services that provide those data so that every service has a generic hyper-media-driven API.

Compared to our approach, OSLO's primary focus is on the interoperability of public services and their data in LOD format. Their goal is to provide a set of tools to create and later validate open data effectively. Our approach is to design a general tool (hence it can be used for any related purpose, such as software development process) for services that use non-RDF formats, such as CSV or XML files, and provide tools to convert them to linked data format.

As noted in the introduction chapter, we also focus on the public sector, but only as one of many use cases. Hence, in general, their approach is better suited for the public sector as their tool may better fit the problem.

6.3 LinkML

LinkML is a general-purpose language and a tool using YAML for modeling schemas that can be converted to various formats such as JSON, CSV, SQL, and RDF schemas, or Python Dataclasses. It also generates human-readable documentation with diagrams and can validate the data in different formats. It is written in Python.

```
classes:
  Person:
    attributes:
      id:
        identifier: true
      full_name:
        required: true
        description: name of the person
      phone:
        pattern: "^[\\d\\(\\)\\-]+$"
      age:
        range: integer
        minimum_value: 0
        maximum_value: 200
```

Figure 6.1: Example of part of the YAML configuration file for LinkML.

The core concepts in their model are classes with properties. Classes then support inheritance, and the model, in general, supports many options for the data types, cardinalities, regular expression patterns, and others.

Nevertheless, the primary input of the framework is the YAML file, where the ontology needs to be defined. This is an entirely different concept than we have, as the ontology already exists, and we model the schemas. The tool seems not to let the user choose what to include to the schema in such granularity as our tool does.

Based on how the framework is used, we suppose that it targets the former use case from the introduction: to create schemas for (micro)service architecture. The lack of UI makes it difficult to use outside this case.

	Enterprise Architect	**OSLO**	**LinkML**	**XCase & eXolutio**	**Dataspecer**
Shared ontology	no	yes	with toolchain	no	yes
Custom schema structure	no	yes	no	yes	yes
Schema extension	n/a	?	n/a	no	planned
Evolution	no	?	no	yes	planned
Custom artifacts	in their DSL	no	by plugin	no	by plugin
User interface	graph	graph	YAML	graph	tree
Pricing	paid	paid due to EA	free	free	free

Table 6.1: Comparison of all the introduced tools with *Dataspecer*. The compared aspects are those that are important for us either by the use case or the requirements from chapters 2 and 3.

Shared ontology - Whether the tool can use a pre-defined ontology that is shared on the Web. The tools that do not support this requirement usually require the user to download it manually or create it directly in it.
Custom schema structure - If it is possible to define your own structure of schema and not have to have it generated automatically from the conceptual model.
Schema extension - Whether the tool supports modifying schemas by creating their copy but keeping the semantic linking between them that could be used for evolution. (see Requirement 12)
Custom artifacts - If it is possible to extend the tool's functionality so custom artifacts can be generated. For example, domain-specific documentation or schemas.
User interface - How the schema is being modeled and with which part of the application the user spends most of the time. *graph* means visual graph representation of the structure whether *tree* is for the bullet-list representation discussed in Requirement 1.

7. Evaluation

To ensure that the framework for modeling and transformation works as desired, we have employed several automated unit tests that covered basic functionality.

We also continuously test the group of transformation generators against each other to quickly find a mistake. For that, we have developed the already mentioned CLI interface, which allows us to create artifacts from a schema, immediately validate test data against the given schema, then transform them into RDF using lifting, store them into a triplestore, use generated SPARQL query to obtain data back to RDF, execute lowering back to given format and again, compare validity against the schema and original data. This is, so far, performed only for XML as we are still working on the other formats.

The application is also in active use to create FOSes (see the introduction chapter) for the Czech government. Our current goal, by which the development was highly affected, is to design a tool that would create them automatically, meeting all requirements, just by modeling schemas from an existing ontology that is being modeled [12] in the Czech Republic.

Currently existing FOSes[1] (or OFNs from *Otevřené formální normy* in Czech) consists of JSON schemas linked to other subschemas and an HTML technical document (similar looking to W3C recommendation documents, for example) in ReSpec[2] containing a description of all concepts used in a schema together with their meaning. Some FOSes also include XML schema, JSON-LD context, and an overview of the RDF structure, as the intent is to map data to RDF, and SPARQL queries. Also, examples of SPARQL queries and data in given formats may be present.

So far, these specifications were made by hand with the help of several scripts for generating structured texts, such as an overview of schema structure. This was, of course, not suitable for wider use, as large parts still needed to be created by hand.

The goal of this chapter is to evaluate the tool in a real-world application and compare the results with the existing FOSes.

7.1 Register of rights and obligations

One well-defined group of FOSes is the Register of rights and obligations (RPP), which currently contains 13 specifications.

During the modeling, a few additional specifications were added that could be later reused by others. Reusing was used extensively, and it was also needed to reuse schemas from other FOSes than the RPP. This was not an issue as all schemas were designed in one instance of the application where individual groups of FOSes were separated by tags. However, in general, even this simple use case already shows that the governmental specifications are interconnected a lot and may be beneficial to have multiple instances of the tool under different institutions, each modeling its own specifications, yet still be able to reuse them.

[1]`https://data.gov.cz/ofn/` (only in Czech)
[2]`https://respec.org/`

Management of schemas The evaluation also shows that there was no data specification that would have more than one schema. In this thesis, we haven't exactly specified how data specifications and data schemas shall be used to create schemas. The mentioned approach was used because each data specification has its documentation, and the modeler can choose which specifications will be reused, as reuse works on the specification level, not the schema level.

Although it may seem that the chosen approach was correct, there is still one schema for specification. In the language of the framework levels, each schema (PSM) had its own PIM.

1. That means that during the evolution in the future, a user would need to evolve all schemas separately, which may cause a problem if one schema is forgotten during the process. On the other hand, evolving the whole set of schemas at once may be challenging as there would be many changes that need to be propagated, and it may be difficult to exclude changes from some schemas.

2. Having a user interface ready for multiple schemas but always using one may be confusing for some users as they can ask a similar question as we do: "When do we need more than one schema?" The answer depends on how we decide the schemas are structured in data specifications. If exactly one schema belongs to a data specification, then we do not need more. But having all schemas under one specification is also a viable approach, especially when designing API for modules, as was introduced in the former example in the introduction chapter. Then, each module would correspond to one data specification having multiple schemas.

Furthermore, the current approach does not support generating artifacts, specifically documentation, for a group of specifications. This is, for example, used by the current documentation of the RPP, as there is one general document that refers to other specifications.

It appears that the problem of schemas belonging to data specifications must be further analyzed as it may not be sufficient enough for larger projects, especially when designing schemas for a government that has multiple branches working more independently yet still needs to share specifications.

Possible solutions would include generalizing a concept of data specification to a project directory, where each schema belongs to one project, and the project may belong to another one (not creating a cycle). However, this would require analyzing how the PIM level shall work, whether each directory shall inherit (see section 4.4) it from the parent and how the evolution shall work.

Schema modeling Regarding the modeling, about half of the schemas contained maximally ten entities, and their purpose was to be referenced from others. The other half had about 20 or 30 entities at most. Most schemas used only standard constructs, such as classes, attributes, associations, and reverse associations. See Figure 8.2 with a screenshot from the application with one of the schemas. About a third of the schemas referenced others. The references did not have cycles, but some of them had lengths of three.

Three schemas required disjunction, specifically the inheritance of classes as specified in Requirement 7, and one schema required the inheritance on the root level. Although the sample is small, it shows that in typical cases, we do not need the disjunction per se, only the inheritance, as usually, when we need to select between two different things, they are usually of the same type.

This specific use case of generating FOSes for the Czech government requires that if the schema represents an array, it shall be an object containing that array. Unfortunately, with the current state of development, this cannot be supported as the creation of non-interpreted classes is not trivial, as it may break the generation of transformation scripts, for example, and hence need to be further analyzed. Currently, those affected schemas can be altered by hand by adding a wrapper object. In the future, this problem can be addressed on two levels.

1. Either introduce a generator that does it automatically, as this is the required behavior for Czech FOSes.

2. Or add support for non-interpreted entities and model the schema with them.

Although the latter approach may seem cleaner yet harder for the user, it may not be correct, as adding the wrapper object may not have the desired semantics. For example, suppose that we would like to reference such schema. Should the wrapper object be present as it is a part of the schema, or do we want to reference the interpreted class instead and set the cardinality correctly?

Another similar requirement is that each interpreted class shall have a *type* attribute with a string value that follows specific pattern rules based on the type of property. This is a very similar problem to the previous one, as the *type* may be considered a domain-specific attribute that is generated automatically.

Documentation Some FOSes had examples that we do not support yet, and it is a subject for future work.

Comparing the generated documentation, our results are missing some schema unrelated info, such as the specification's author or European Union logo. In general, this shall be solved by introducing a new documentation generator that works similarly but adds those missing information and sets the structure as required.

Our tool successfully generated descriptions for the conceptual model with diagrams (which some FOSes do not have) and the documentation of the schemas. The schema documentation, however, is a little harder to read. Therefore we need to focus on improving this as well.

In general, because the original documentation was hard to make by hand and various scripts were used to generate it, it shouldn't be challenging to achieve almost the same result by modifying the generator accordingly.

Conclusion

In this thesis, we have analyzed, formally described, and implemented a newly developed framework and a tool for schema modeling and management based on Model-Driven Architecture (MDA) and previously developed tools *XCase* and *eXolutio*.

We have implemented the core framework functionality of complete modeling of schemas from an ontology with support of inheritance of entities and disjunction. We have created an easy-to-use user interface that provides all the concepts for the modeling and management of schemas in specifications and artifact generation.

Next, this thesis laid the foundations for future work in this area, which was described and analyzed in chapters 3 and 4, such as the use of non-interpreted entities, evolution, and inheritance of schemas or use of recommendation systems in modeling. These topics were complex to be analyzed alongside the core functionality of the tool and would require separate work, but they were necessary to be considered to implement the framework properly.

The tool is constantly used for modeling recommendations for publishing open data of public institutions and the government of the Czech Republic.

Dataspecer is open-source and developed on GitHub. Technical documentation is part of the repository. User documentation with running instance and additional information about the project, and the link to the repository is available at the project website `dataspecer.com`.

Bibliography

[1] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan De Lara. Recommender systems in model-driven engineering. *Software and Systems Modeling*, 21(1):249–280, 2022.

[2] Newton Calegari, Caroline Burle, and Bernadette Farias Loscio. Data on the web best practices. W3C recommendation, W3C, January 2017. https://www.w3.org/TR/2017/REC-dwbp-20170131/.

[3] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. *SIGMOD Rec.*, 28(2):431–442, jun 1999.

[4] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. RML: a generic language for integrated RDF mappings of heterogeneous data. In *Ldow*, 2014.

[5] Jérôme Euzenat and Pavel Shvaiko. *Ontology Matching.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[6] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures.* University of California, Irvine, 2000.

[7] Giancarlo Guizzardi, Alessander Botti Benevides, Claudenir M. Fonseca, Daniele Porello, João Paulo A. Almeida, and Tiago Prince Sales. UFO: Unified Foundational Ontology. *Applied Ontology*, Pre-press(Pre-press):1–44, 2021.

[8] Stuart Kent. Model driven engineering. In *International conference on integrated formal methods*, pages 286–298. Springer, 2002.

[9] Meike Klettke and Holger Meyer. Xml and object-relational database systems enhancing structural mappings based on statistics. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Dan Suciu, and Gottfried Vossen, editors, *The World Wide Web and Databases*, pages 151–170, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[10] Jakub Klímek, Lukáš Kopenec, Pavel Loupal, and Jakub Malý. XCase - A Tool for Conceptual XML Data Modeling. In *Advances in Databases and Information Systems, Associated Workshops and Doctoral Consortium of the 13th East European Conference, ADBIS 2009, Riga, Latvia, September 7-10, 2009. Revised Selected Papers*, volume 5968 of *LNCS*, pages 96–103. Springer, 2009.

[11] Jakub Klímek, Jakub Malý, Martin Nečaský, and Irena Holubová. eXolutio: Methodology for Design and Evolution of XML Schemas Using Conceptual Modeling. *Informatica*, 26(3):453–472, 2015.

[12] Petr Křemen and Martin Nečaský. Improving discoverability of open government data with rich metadata descriptions using semantic government vocabulary. *Journal of Web Semantics*, 55:1–20, 2019.

[13] Sam Newman. *Monolith to Microservices.*

[14] Martin Nečaský. XSEM: a conceptual model for XML. 2007.

[15] Martin Nečaský, Jakub Klímek, Jakub Malý, and Irena Mlýnková. Evolution and change management of XML-based systems. *Journal of Systems and Software*, 85(3):683–707, 2012.

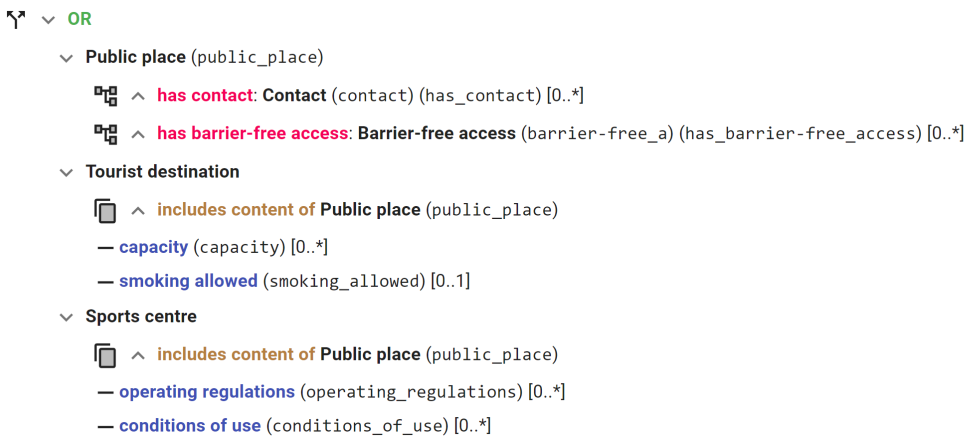
[16] Martin Nečaský, Irena Mlýnková, Jakub Klímek, and Jakub Malý. When conceptual model meets grammar: A dual approach to XML data modeling. *Data & Knowledge Engineering*, 72:1–30, 2012.

[17] Jon Siegel. Developing in OMG's Model-Driven Architecture. page 12.

[18] Richard Soley. Model Driven Architecture. *Model Driven Architecture*, page 12.

[19] Štěpán Stenchlák, Martin Nečaský, Petr Škoda, and Jakub Klímek. Dataspecer: A model-driven approach to managing data specifications. In *European Semantic Web Conference*. Springer, 2022.

8. Attachments



OR

Public place (public_place)

has contact: Contact (contact) (has_contact) [0..*]

has barrier-free access: Barrier-free access (barrier-free_a) (has_barrier-free_access) [0..*]

Tourist destination

includes content of Public place (public_place)

— capacity (capacity) [0..*]

— smoking allowed (smoking_allowed) [0..1]

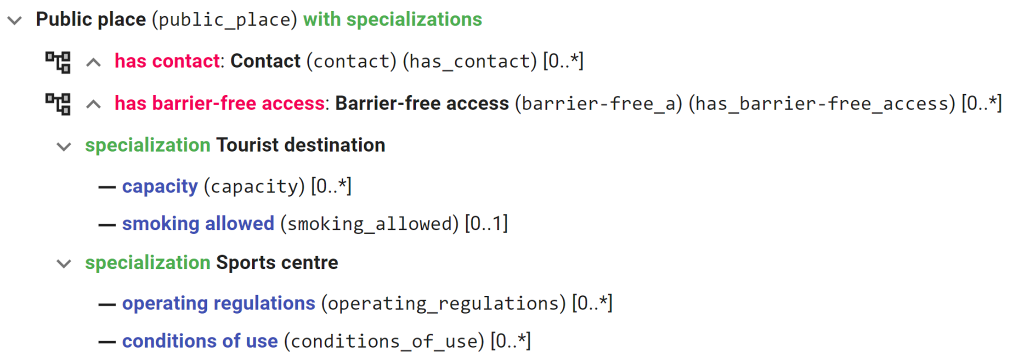
Sports centre

includes content of Public place (public_place)

— operating regulations (operating_regulations) [0..*]

— conditions of use (conditions_of_use) [0..*]

(a) General schema with OR in the root of the schema and several includes.



Public place (public_place) with specializations

has contact: Contact (contact) (has_contact) [0..*]

has barrier-free access: Barrier-free access (barrier-free_a) (has_barrier-free_access) [0..*]

specialization Tourist destination

— capacity (capacity) [0..*]

— smoking allowed (smoking_allowed) [0..1]

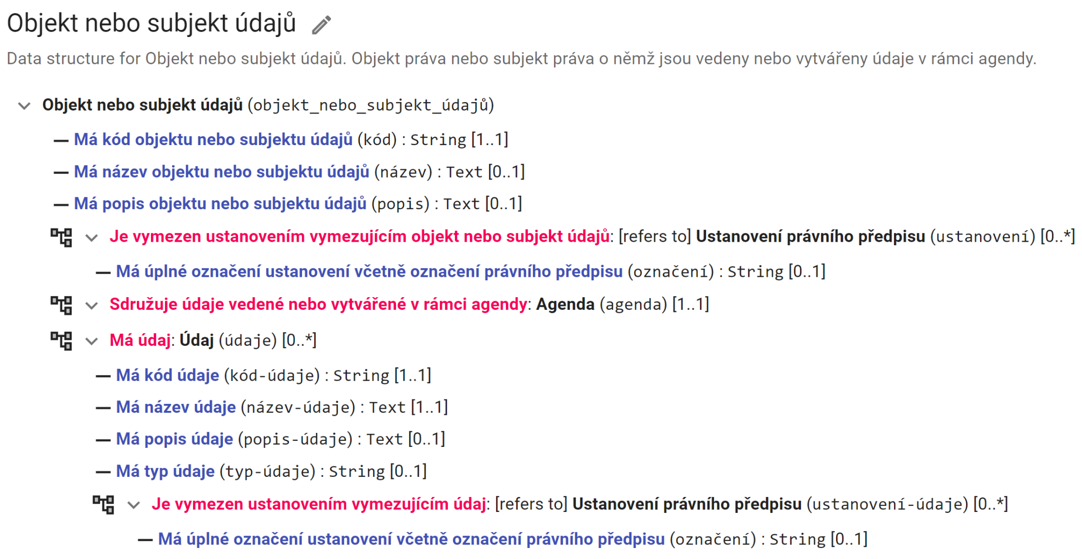
specialization Sports centre

— operating regulations (operating_regulations) [0..*]

— conditions of use (conditions_of_use) [0..*]

(b) Same general schema visualized as one base class with specializations.

Figure 8.1: Screenshots from the tool comparing a general schema of a Public place with two specializations according to Requirement 7. The variants show the plain view, consisting of OR and include, and a user-friendly view hiding those constructs. The example shows one of the FOSes that we are trying to model. For the purpose of this example, the schema was simplified, some classes were contracted, and labels were translated to English.


Objekt nebo subjekt údajů

Data structure for Objekt nebo subjekt údajů. Objekt práva nebo subjekt práva o němž jsou vedeny nebo vytvářeny údaje v rámci agendy.

```
Objekt nebo subjekt údajů (objekt_nebo_subjekt_údajů)
  — Má kód objektu nebo subjektu údajů (kód) : String [1..1]
  — Má název objektu nebo subjektu údajů (název) : Text [0..1]
  — Má popis objektu nebo subjektu údajů (popis) : Text [0..1]
  Je vymezen ustanovením vymezujícím objekt nebo subjekt údajů: [refers to] Ustanovení právního předpisu (ustanovení) [0..*]
    — Má úplné označení ustanovení včetně označení právního předpisu (označení) : String [0..1]
  Sdružuje údaje vedené nebo vytvářené v rámci agendy: Agenda (agenda) [1..1]
  Má údaj: Údaj (údaje) [0..*]
    — Má kód údaje (kód-údaje) : String [1..1]
    — Má název údaje (název-údaje) : Text [1..1]
    — Má popis údaje (popis-údaje) : Text [0..1]
    — Má typ údaje (typ-údaje) : String [0..1]
    Je vymezen ustanovením vymezujícím údaj: [refers to] Ustanovení právního předpisu (ustanovení-údaje) [0..*]
      — Má úplné označení ustanovení včetně označení právního předpisu (označení) : String [0..1]
```

Figure 8.2: A real-life example of a general schema of one of the RPP FOSes, that were described in section 7.1. You can see that the schema consists of only a few associations and attributes. There are two references to the other schema denoted by *[refers to]* text.