Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Data Model Creation with MetaConfigurator

Felix Neubauer

Course of Study:	Softwaretechnik
Examiner:	Jun.-Prof. Dr. Benjamin Uekermann
Supervisor:	Prof. Dr. Jürgen Pleiss
Commenced:	April 1, 2024
Completed:	September 30, 2024

Abstract

In both research and industry, significant effort is devoted to the creation of standardized data models that ensure data adheres to a specific structure, enabling the development and use of common tools. These models (also called schemas), enable data validation and facilitate collaboration by making data interoperable across various systems. Tools can assist in the creation and maintenance of data models. One such tool is MetaConfigurator, a schema editor and form generator for JSON schema and for JSON/YAML documents. It offers a unified interface that combines a traditional text editor with a graphical user interface (GUI), supporting advanced schema features such as conditions and constraints. In this work, MetaConfigurator is viewed from the perspective of three exemplary real-world use case in fields such as biochemistry and ontology management. Multiple improvements and functionalities are designed and implemented to further assist the user: 1) A more user-friendly schema editor, distinguishing between an easy and an advanced mode based on a novel *meta schema builder* approach; 2) A CSV import feature for seamless data transition from Excel to the JSON format with schema inference; 3) Snapshot sharing for effortless collaboration; 4) Ontology integration for auto-completion of URIs; and 5) A novel graphical diagram-like schema view for visual schema manipulation. These new functionalities are then applied to the real-world use cases, demonstrating the practical utility and improved accessibility of MetaConfigurator.

Kurzfassung

Sowohl in der Forschung als auch in der Industrie wird erheblicher Aufwand in die Erstellung standardisierter Datenmodelle investiert. Diese stellen sicher, dass Daten einer bestimmten Struktur folgen und ermöglichen die Entwicklung und Nutzung gemeinsamer Werkzeuge. Diese Modelle (auch Schemata genannt) ermöglichen die Validierung von Daten und erleichtern die Zusammenarbeit, indem sie die Interoperabilität von Daten über verschiedene Systeme hinweg gewährleisten. Tools können bei der Erstellung und Pflege von Datenmodellen unterstützen. Eines dieser Tools ist MetaConfigurator, ein Schema-Editor und Formulargenerator für JSON-Schemata sowie für JSON/YAML-Dokumente. Es bietet eine einheitliche Benutzeroberfläche, die einen traditionellen Texteditor mit einer grafischen Benutzeroberfläche (GUI) kombiniert und erweiterte Schemafunktionen wie Bedingungen und Einschränkungen unterstützt. In dieser Arbeit wird MetaConfigurator aus der Perspektive von drei exemplarischen realen Anwendungsfällen in Bereichen wie der Biochemie und dem Ontologiemanagement betrachtet. Mehrere Verbesserungen und neue Funktionen werden entworfen und implementiert, um den Benutzer weiter zu unterstützen: 1) Unterscheidung zwischen einem einfachen und einem fortgeschrittenen Modus beim Schema-Editor, auf Grundlage eines neuartigen *Meta Schema Builder*-Ansatzes; 2) Eine CSV-Importfunktion für einen nahtlosen Übergang von Excel Dateien zu JSON, mit Schema-Inferenz; 3) Snapshot-Sharing für eine mühelose Zusammenarbeit; 4) Eine Ontologie-Integration für die automatische Vervollständigung von URIs; und 5) Eine neuartige grafische, diagramm-ähnliche Schema-Ansicht für die visuelle Manipulation von Schemata. Diese neuen Funktionen werden anschließend auf die realen Anwendungsfälle angewendet, um den praktischen Nutzen und die verbesserte Zugänglichkeit von MetaConfigurator zu demonstrieren.

Contents

Acronyms

API Application Programming Interface. 41

AWS Amazon Web Services. 20

CSV Comma-Separated Values. 9

DTD Document Type Definition. 13

FAIR Findable, Accessible, Interoperable, and Re-usable. 28

FOAF Friend Of A Friend. 23

GUI Graphical User Interface. 10

HTML Hypertext Markup Language. 40

IDE Integrated Development Enviroment. 19

IRI Internationalized Resource Identifier. 23

JSON JavaScript Object Notation. 5

JSON-LD JSON for Linked Data. 27

JTD JSON Type Definition. 13

LLM Large Language Model. 74

NFDI Nationale Forschungsdaten Infrastruktur. 9

NOMAD Novel Materials Discovery. 9

OPC UA Open Platform Communication - Unified Architecture. 28

OWL Web Ontology Language. 22, 27

RDF Resource Description Framework. 22

RDFS RDF Schema. 22, 25

RDM Research Data Management. 9

REST Representational State Transfer. 41

sdRDM Software-driven RDM. 28

SHACL Shapes Constraint Language. 22, 26

SiLA Standardization in Lab Automation. 28

SPARQL SPARQL Protocol and RDF Query Language. 25

SQL Structured Query Language. 25

Turtle Terse RDF Triple Language. 23

UI User Interface. 15

UML Unified Modeling Language. 11

URI Uniform Resource Identifier. 11, 23

URL Uniform Resource Locator. 40

UUID Universally Unique Identifier. 42

XSD XML Schema Definition. 13

1 Introduction

In both research and industry, data has become increasingly important for driving innovation and generating new insights [75, 86]. To facilitate interoperability across systems, structured *data formats*, such as *JSON* [30], *YAML* [12], *XML* [103], and *Comma-Separated Values (CSV)* are used. Data in these formats differs from natural language by following a strict *syntax* [6, 26], making it *parsable* and *readable* by a machine in a *deterministic* manner. Data formats define the *syntax* of data, but neither the *semantics* (meaning) [58] of it, nor which properties are required or which constraints they need to satisfy. On top of a data format, a *data model* (schema) can be introduced, which specifies required properties and constraints and can imply semantics (for example through descriptions). Listings 1.1-1.4 illustrate the differences between a data format and a data model.

```
{
  "firstName": "Alex",
  "lastName": "Smith",
  "age": 28
}
```

Listing 1.1: Document in valid JSON syntax, which also satisfies the data model defined in listing 1.4.

```
{
  "firstName": "Alex",
  "lastName": "Smith"
  "age": 28
}
```

Listing 1.2: Document that does not follow JSON syntax because of a missing comma.

```
{
  "firstName": 28,
  "lastName": "Smith",
  "age": "Alex"
}
```

Listing 1.3: Document in valid JSON syntax, which does not satisfy the data model defined in listing 1.4.

This thesis is written with Research Data Management (RDM) and standardization initiatives, such as Novel Materials Discovery (NOMAD) [34] and Nationale Forschungsdaten Infrastruktur (NFDI) [49] in mind. However, the idea of specifying data models is generic and can be applied in any domain. Industry also benefits from standardized data models [40].

```
{
"properties": {
    "firstName": {
      "type": "string",
    },
    "lastName": {
      "type": "string",
    },
    "age": {
      "type": "integer",
    }
  }
}
```

Listing 1.4: Exemplary data model.

A data model (also called schema) allows communicating the data structure with others, validating data and developing common tools and techniques, which manipulate or analyze the data. However, despite their clear advantages, the creation and adoption of standardized data models are often neglected. Many industries and research fields still rely on ad-hoc data structures, typically found in tools, such as Excel (see section 3.3), which, while flexible, lack the rigor required for collaboration across different systems and domains [50]. This lack of standardization can lead to inconsistencies, inefficiencies, and data loss when information needs to be exchanged or integrated across systems [100]. Even when organizations recognize the need for data models, they frequently develop custom models that are not compatible with those used by other organizations. This results in fragmentation and siloed data ecosystems, where valuable information cannot be easily reused or shared [59, 77]. The challenge is twofold: first, creating data models that can serve as common ground for multiple stakeholders requires significant expertise. Second, manually editing or updating these models is a time-consuming and error-prone process, particularly for users who are not familiar with schema languages. This creates a clear need for tools that simplify the creation and management of data models, ensuring they can be widely adopted and maintained with minimal technical overhead.

To address this challenge, *MetaConfigurator* was developed [72] as an open-source tool[1] designed to help users create and edit data models for data in the JSON or YAML format. *MetaConfigurator* offers a unified interface that combines a traditional text editor with a Graphical User Interface (GUI) editor for additional assistance. It supports advanced schema features, such as conditions, constraints, and composition, enabling the creation of comprehensive and robust data models. The tool uses the schema language *JSON Schema* [52, 79]. In a previous work, we had compared different schema languages by *expressiveness*, *popularity* and *tooling support* and found JSON schema to be the most suitable one[2]. It is very expressive, popular and is supported by all major programming languages. Therefore, data models developed in JSON schema can easily be shared with and re-used by others.

[1] https://github.com/MetaConfigurator/meta-configurator

[2] https://github.com/MetaConfigurator/meta-configurator/paper/paper_main_extended.pdf accessed 2024/04/27

Discussions with researchers and practitioners from various fields, including biochemistry and developers of the multi-physics coupling library *preCICE* [27], highlighted interest in additional functionalities that would further improve the tool's utility and simplify schema editing. Within this thesis these requests are addressed and new functionalities designed and implemented. Because of the expressiveness of JSON schema, the schema editing in *MetaConfigurator* can be complicated for beginners. To lower the entry barrier and hide complexity from the user a distinction between an easy and an advanced mode is introduced. In the easy mode, more advanced schema features are hidden from the user. This achieved using a novel *meta schema builder* approach. Also, a novel graphical schema view is introduced, which visualizes schemas in a way similar to a Unified Modeling Language (UML) class diagram and allows for graphical editing of the schema. Furthermore, the Biochemistry researchers from *University of Stuttgart* store a lot of their research data in Excel files. To enable for a smooth transition to the JSON format, a CSV import functionality is designed and implemented. This allows the researchers to convert their data into JSON with a few clicks and also automatically infer the schema instead of having to perform these tasks by hand. Additionally, because there already exist different ontologies in the field of Biochemistry and the researchers are interested in linking their data models with it, a novel ontology integration is added to *MetaConfigurator*, providing auto-completion for existing ontology Uniform Resource Identifiers (URIs). When research data is linked to a knowledge graph, additional information can be inferred. The research data itself could also be embedded into a knowledge graph. Finally, to foster collaboration and make it easy for users to share their data or schema, a snapshot sharing functionality is developed.

Besides the Biochemistry researchers, developers of the *preCICE* coupling library also want to make use of *MetaConfigurator* for creating a data model. Their standardization process will involve many different stakeholders and can last a year or longer. Having a user-friendly schema editor and form generator would assist them in the process and also help them communicate the data model to the *preCICE* users. They had many different suggestions and wishes for smaller adaptations and improvements to the user interface of *MetaConfigurator*, which are also implemented and discussed in this thesis.

The thesis is structured as follows: Chapter 2 presents related work, including an introduction to JSON, JSON Schema, the original version of *MetaConfigurator*, ontologies and schema visualization techniques. Chapter 3 details the design and implementation of the new functionalities. Chapter 4 demonstrates the application of these features in real-world use cases, particularly in fields such as biochemistry and ontology management. Finally, Chapter 5 summarizes the contributions and discusses potential future work.

2 Background and Related Work

This chapter introduces related work, which forms the basis for this thesis and is required for understanding of parts of the thesis. Parts of the related work sections about JSON Schema and MetaConfigurator were taken over and adapted from our previous paper [72].

2.1 JSON Schema

JSON is a common data-interchange format for exchanging data with web services, but also for storing documents in NoSQL databases, such as MongoDB [68]. *Schema languages* are formal languages that specify the structure, constraints, and relationships of data, for example, in a database or structured data formats. Schema languages exist for *JSON* (*JSON Schema* [52, 79], *Apache Avro* [63], *JSON Type Definition (JTD)* [21], *TypeSchema* [55], etc.), but also for other data formats, such as *XML* (*Document Type Definition (DTD)* [14], *XML Schema Definition (XSD)* [37], and others [64, 69]). In our initial (extended) paper about *MetaConfigurator*[1], we evaluated 8 different schema languages, measuring their *practical usage* (*popularity*, *tool support*, *library support*) and *expressiveness*. We found *JSON Schema* [52, 79] to be the one with most *practical usage* and *expressiveness*, which is why we based *MetaConfigurator* on it.

Listing 2.1 shows an example of a JSON schema, and listing 2.2 shows an example of a JSON document that conforms to the schema. The schema defines the structure of a *Person* document, having the required properties *firstName*, *lastName* and *age*. Additionally, it has an optional recursive property *bestFriend*, which is also of type *Person*.

JSON schema has evolved to being the de-facto standard schema language for JSON documents [4]. Schemas for many popular configuration file types exist. *JSON schema store*[2] is a website that provides over 600 JSON schema files for various use cases. The supported file types include, for example, Docker compose or OpenAPI files. JSON schema and other schema languages for JSON can also be applied to YAML, as JSON and YAML documents have a similar structure (JSON is a subset of YAML). Some syntactical details of YAML can, however, not be expressed with JSON schema.

[1] https://github.com/MetaConfigurator/meta-configurator/paper/paper_main_extended.pdf accessed 2024/09/27
[2] https://www.schemastore.org/json/, accessed 2024/09/27

```json
{
  "$id": "https://example.org/person.schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Person",
  "type": "object",
  "required": ["firstName", "lastName", "age"],
  "properties": {
    "firstName": {
      "type": "string",
      "description": "first name."
    },
    "lastName": {
      "type": "string",
      "description": "last name."
    },
    "age": {
      "description": "Age",
      "type": "integer",
      "minimum": 0
    },
    "bestFriend": {
      "$ref": "#"
    }
  }
}
```

Listing 2.1: JSON schema example.

```json
{
  "firstName": "Alex",
  "lastName": "Smith",
  "age": 28
}
```

Listing 2.2: JSON example for the schema in listing 2.1.

2.1.1 Schema-based Form Generation

The idea of generating a GUI from a schema is not new. Early works in this area propose generating forms from XML schemas [38, 56, 60] or entity-relationship diagrams [7]. There exist various approaches that generate web forms from the more modern format, JSON schema, e.g., *React JSON Schema Form*[3], *Angular Schema Form*[4], *Vue Form Generator*[5], *JSON Forms*[6], *JSON Editor Online*[7], and *JSON Form*[8].

[3] https://github.com/rjsf-team/react-jsonschema-form, accessed 2024/09/24.
[4] https://github.com/json-schema-form/angular-schema-form, accessed 2024/09/24.
[5] https://github.com/vue-generators/vue-form-generator, accessed 2024/09/24.
[6] https://jsonforms.io, accessed 2024/09/24.
[7] https://jsoneditoronline.org, accessed 2024/09/24.
[8] https://github.com/jsonform, accessed 2024/09/24.

Such forms can assist the user in a multitude of ways, such as by tooltips, auto-completion, and dropdown menus. By inherently adhering to the schema structure, editing data with such GUIs significantly reduces configuration mistakes caused by the user. The generated forms usually have a specific component for each type of data, e.g., a text field for strings.

2.1.2 Schema Editors

There exist several so-called schema editors, which are tools for creating and editing schemas that are text-based or graphical (or both). *JSON Editor Online*[9] is a web-based editor for JSON schemas and JSON documents. It divides the editor into two parts, where one part can be used to edit the schema and the other part can be used to edit a JSON document, which is validated against the schema. The editor provides various features, such as syntax highlighting and highlighting of validation errors. However, the features of the editor are limited. For example, it does not provide any assistance for the user, such as tooltips or auto-completion. For new documents, it does not show any properties of the schema, so the user has to know the schema beforehand

There also exists a variety of schema editors that are paid software, such as *Altova XMLSpy*[10], *Liquid Studio*[11], *XML ValidatorBuddy*[12], *JSONBuddy*[13], *XMLBlueprint*[14], and *Oxygen XML Editor*[15]. Those are editors for XML or JSON schema, mostly with a combination of text-based and graphical views. These tools are not web-based and not open-source. Furthermore, they do not focus on editing a JSON document based on a schema, but rather only on editing the schema itself.

2.1.3 Adamant

Adamant[16] is a JSON Schema-based form generator and schema editor specifically designed for scientific data [90]. It generates a GUI from a JSON schema, allows editing and creating JSON schema documents, and differentiates between a schema edit mode and a data edit mode. A noteworthy feature is that it supports the extraction of units from the description of a field, which is helpful for scientific data.

Besides the frontend, Adamant also provides a backend that allows the integration into other tools as well as storing and reusing schemas. Adamants User Interface (UI)-based schema editor is intuitive and user-friendly, even for users who are not familiar with JSON schema. However, Adamant does not provide a text based editor as an alternative to the GUI and does not support various JSON schema keywords. For example, it is not possible to restrict strings to a certain regular expression and it does not support the keyword `oneOf`, which many schemas use [4].

[9] https://jsoneditoronline.org, accessed 2024/09/24.
[10] https://www.altova.com/xmlspy-xml-editor, accessed 2024/09/24.
[11] https://www.liquid-technologies.com/json-schema-editor, accessed 2024/09/24.
[12] https://www.xml-buddy.com/, accessed 2024/09/24.
[13] https://www.json-buddy.com/, accessed 2024/09/24.
[14] https://www.xmlblueprint.com, accessed 2024/09/24.
[15] https://www.oxygenxml.com, accessed 2024/09/24.
[16] Current version of Adamant as of writing this thesis: Adamant v1.2.0

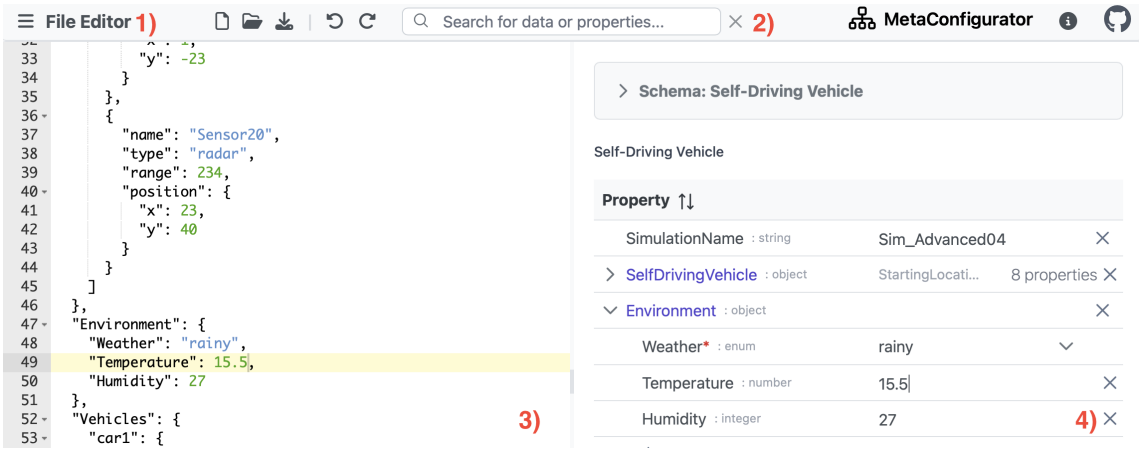

Figure 2.1: UI of the *MetaConfigurator* file editor view. Different components highlighted in red: 1) button to switch to other view (e.g. to Schema Editor view), 2) Toolbar with various functionality, 3) Text editor panel, 4) GUI panel.

2.2 MetaConfigurator

MetaConfigurator [72] is a general-purpose form generator and schema editor that is not bound to a specific domain. It uses a modular and extensible architecture that supports different data formats (e.g., JSON and YAML) and different ways to present and edit the data (e.g., a text editor and a GUI editor). The schema editor and the UI for editing data are the same, as the schema itself is treated as a configuration file. *MetaConfigurator* is a client-side web application, which means that it runs in the browser of the user and does not require a server.

2.2.1 User Interface

MetaConfigurator has three distinct views:

1. File editor (figure 2.1): In this view, the user can modify their structured data file, based on a schema.
2. Schema editor (figure 2.2): In this view, the user can modify their schema.
3. Settings: In this view, the user can adjust the parameters of the tool.

The UI of each view is divided into two main panels: the *text editor* (on the left) and the *GUI editor* (on the right). In the *text editor*, the user can modify their data by hand, the same way as in a regular text editor. Features, such as syntax highlighting and schema validation, assist the user. In the *GUI panel*, the user can modify their data with the help of a GUI. The GUI is based on the JSON schema file that the user provides. This design combines the benefits of both a text editor with the benefits of a GUI. A text editor is efficient for many tasks and more suited for users with a technical understanding of the data structure, while a GUI enables users without deep technical understanding to work with the data. Nevertheless, a GUI also simplifies the editing process for expert users. As a schema is a structured data file itself, it is treated as such, and the tool offers assistance accordingly. The user can edit the schema in the same way as they can edit their data. An

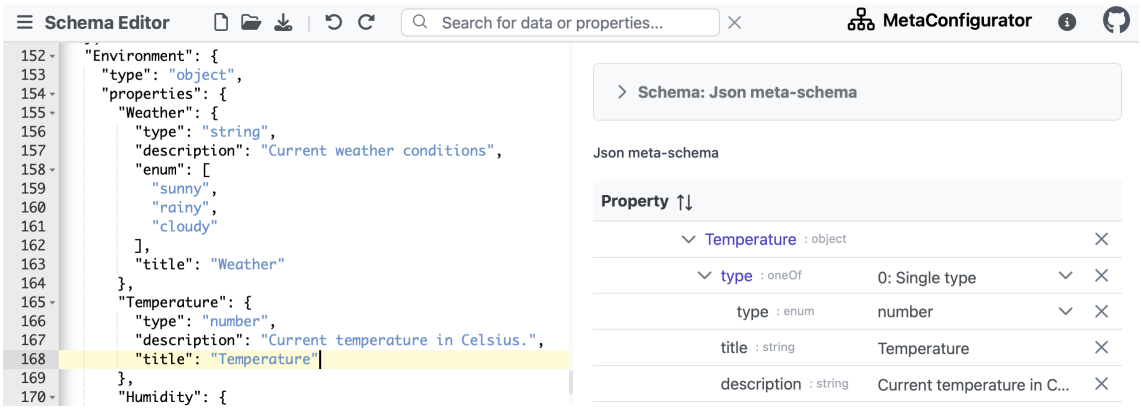

Figure 2.2: The *MetaConfigurator* schema editor view. The schema editor is based on the same UI as the file editor.

Table 2.1: File data and schema for the different views.

Mode	Data	Schema
File editor	User data	User schema
Schema editor	User schema	JSON Schema meta-schema
Settings	Settings data	Settings schema

adapted version of the JSON schema meta-schema is used to generate the UI for the schema editor. Because of this design, the schema editor itself is generated automatically and, therefore, supports all JSON schema keywords. Whenever the user edits a structured data file using the tool, they do so using some underlying schema. Even the settings of the tool are treated as a structured data file, for which there is an underlying settings schema. Table 2.1 shows which data and schema the tool uses in the different views/modes.

To represent the schema in the GUI editor, preprocessing of the schema is performed. A one-time preprocessing step is performed when loading the schema, performing small manipulations, such as inducing titles and inferring the types of enums. Secondly, internal lazy preprocessing happens at every layer of the schema tree, primarily resolving references and merging `allOf` sub-schemas. The third preprocessing step is calculated every time the data changes, processing conditionals (which are data-dependent).

The schema editor view, mirroring the file editor structure, employs a custom JSON schema meta-schema instead of the user-provided schema to generate the GUI panel. It is an adapted version of the official meta schema with dynamic references and anchors replaced by non-dynamic references and other changes, intended to yield in a intuitive GUI editor. To further reduce the number of fields shown to the user, a mechanism is introduced to hide fields that are not necessary for the basic usage of the schema. This is implemented by introducing a custom keyword `advanced` to the meta-schema. Fields marked as `advanced` are hid behind an expandable `advanced` category in the GUI.

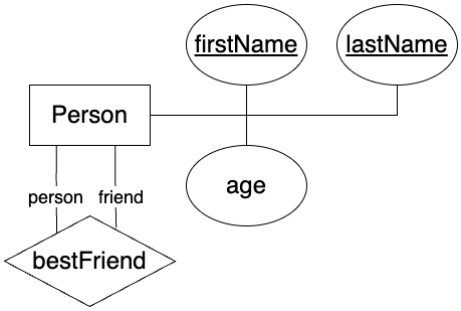

Figure 2.3: Example entity relationship diagram in Chen notation.

During development, a user study was conducted with five participants. Among the five participants, four had a background in research data management. None of the participants had extensive experience with JSON schema, but all of them had experience with JSON and YAML files. The participants were shortly introduced to *MetaConfigurator*, and tasks were given (retrieving data from a file, modifying a file, and editing the schema). 90% of the tasks given to the participants were solved without the need for any help. The remaining 10% of tasks were solved after we provided the participant with minor hints regarding the tool. All participants noted that they can imagine using *MetaConfigurator* in practice for their own work. Three participants highlighted the *intuitiveness* of the approach. Four participants explicitly described the tool as *useful* or *helpful*. The separation into text editor and GUI editor was also commented on positively by four participants. Besides the mainly positive feedback, two participants commented on the schema editor being more complicated than the file editor.

2.3 Schema Visualization

This section describes existing work related to visualizing data models and schemas.

2.3.1 Entity Relationship Diagrams

The entity relationship model is a conceptual data model and was proposed by Chen [24, 25] in 1976, focusing on relational database design. The three core components of the model are *entities*, *relationships* and *attributes*. Figure 2.3 shows an example of the JSON schema from listing 2.1 modelled as a relational data model, in Chen notation. The `Person` *entity* is depicted with a rectangle, *attributes* with a circle and *relationships* with a diamond shape. In the example, the `firstName` and `lastName` attributes make up the *primary key* for the entity in the database, which is why they are underlined. Besides the diagrammatic notation described by Chen, others have been proposed, such as the *Bachmann* notation and notations that allow *n-ary* relationships [98]. Song et al. [92] compare different notations.

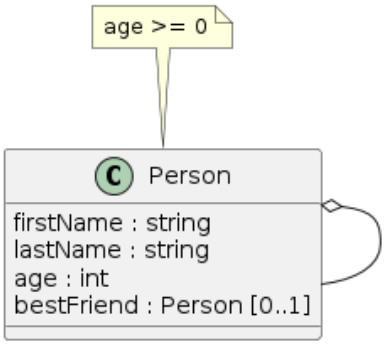

Figure 2.4: Example UML class diagram.

2.3.2 UML Diagrams

UML [36, 80, 84] is a modeling language to standardize the visualization of the design of a system. UML specifies different types of view (*use-case*, *logical*, *implementation*, *process* and *deployment* views) and different types of diagrams (*class diagrams*, *object diagrams*, *state machines*, *activity diagrams*, *interaction diagrams* and others). Figure 2.4[17] shows a UML diagram that models the `Person` class from the JSON schema from listing 2.1.

2.3.3 JSON Schema Visualization

A JSON document has a tree-like structure. After resolving references, it resembles a graph and might contain cycles. There exist different tools for visualizing arbitrary JSON documents, such as *JSON CRACK*[18] or *PlantUML JSON visualization*[19]. Figure 2.5 shows the example JSON schema from listing 2.1 visualized with *JSON CRACK*.

JSON schema documents are a subset of all JSON documents and have a more specialized shape. There also exist tools for visualizing JSON schemas, such as *Schema Visualizer*[20], *JSON Schema Viewer*[21] and the *Jetbrains*[22] Integrated Development Enviroment (IDE) plugin *JSON Schema Visualizer/Editor*[23].

Furthermore, there exist techniques for generating UML diagrams [84] from a JSON schema, such as the *Eclipse*[24] IDE plugin *JsonSchema-to-uml*[25]. In the plugin, each JSON schema element is represented by a UML class. Properties of JSON Schema elements represent the properties of a UML class. Primitive properties are modeled as attributes and enum properties are modeled with an enumeration. If a property is of type object or refers to another element (using `$ref`), an

[17] created with http://www.plantuml.com/plantuml/uml/, accessed 2024/09/24.
[18] https://github.com/AykutSarac/jsoncrack.com, accessed 2024/09/24.
[19] https://plantuml.com/de/json, accessed 2024/09/24.
[20] https://github.com/shamilnabiyev/schema-visualizer, accessed 2024/09/24.
[21] https://github.com/jlblcc/json-schema-viewer, accessed 2024/09/24.
[22] http://jetbrains.com, accessed 2024/09/27.
[23] https://plugins.jetbrains.com/plugin/23554-json-schema-visualizer-editor, accessed 2024/09/27.
[24] https://eclipseide.org/, accessed 2024/09/27.
[25] https://github.com/SOM-Research/JsonSchema-to-uml, accessed 2024/09/27.

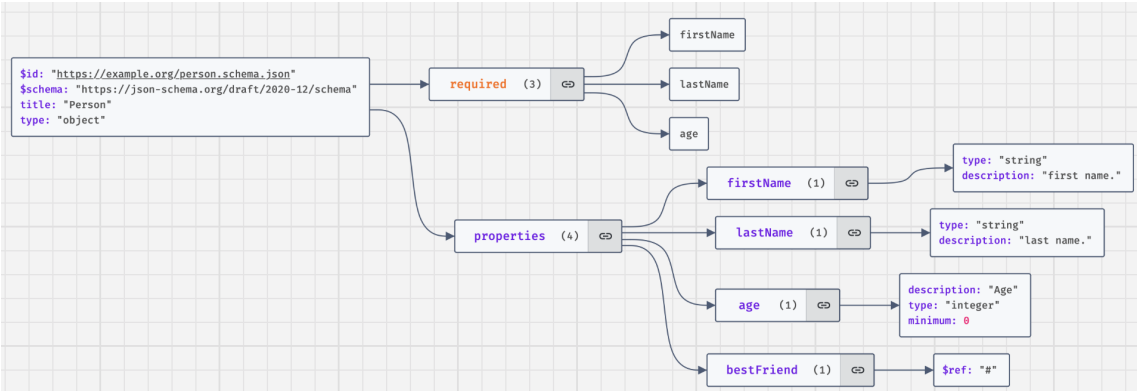

Figure 2.5: JSON schema defined in listing 2.1 visualized by *JSON CRACK*.

association is created in the UML class. The JSON schema keywords `allOf`, `oneOf` and `anyOf` lead to the creation of a hierarchy. After having generated a UML diagram in XML data format, other tooling could be used for visualizing it. The plugin does not represent conditionals and some other advanced JSON schema keywords in UML.

There is no exact mapping from advanced JSON schema keywords to its representation in UML. For example, UML specifies *association* (objects of two classes can know about each other) *composition* (class A is an integral part of class B), *aggregation* (somewhere in-between the previous two) and *inheritance* (attributes of class A are inherited by class B), but none of these matches completely with the JSON schema `oneOf` keyword. `oneOf` is comparable to conditional inheritance, where a class inherits from only one and exactly one other class of a set of classes. In UML, this could be modeled using *constraints*, which can be written in natural language (e.g., English), mathematical notations or other forms. Analogously, the JSON schema conditionals `if`, `then` and `else` can be modeled using UML constraints.

2.3.4 TypeScript Libraries

To visualize knowledge graphs, there exist several libraries for TypeScript. For example *Graph Explorer*[26] from *Amazon Web Services (AWS)*[27] for visualization of RDF data, or *neovis.js*[28] for visualization of data from Neo4j[29] graphs. Also, there exist different libraries for drawing custom graphs, with specified nodes and edges, such as *REAGRAPH*[30] for *React*[31], *v-network-graph*[32] for *Vue.js*[33] and *Cytoscape*[34]. To visualize flow diagrams, there are *Vue Flow*[35] and the graph

[26] https://github.com/aws/graph-explorer, accessed 2024/05/04.
[27] https://aws.amazon.com, accessed 2024/05/04.
[28] https://github.com/neo4j-contrib/neovis.js, accessed 2024/05/04.
[29] https://neo4j.com, accessed 2024/05/04.
[30] https://github.com/reaviz/reagraph, accessed 2024/05/04.
[31] https://react.dev, accessed 2024/05/04.
[32] https://github.com/dash14/v-network-graph, accessed 2024/05/04.
[33] https://vuejs.org, accessed 2024/05/04.
[34] https://js.cytoscape.org, accessed 2024/05/04.
[35] https://vuejs.orghttps://vuejs.org, accessed 2024/05/04.

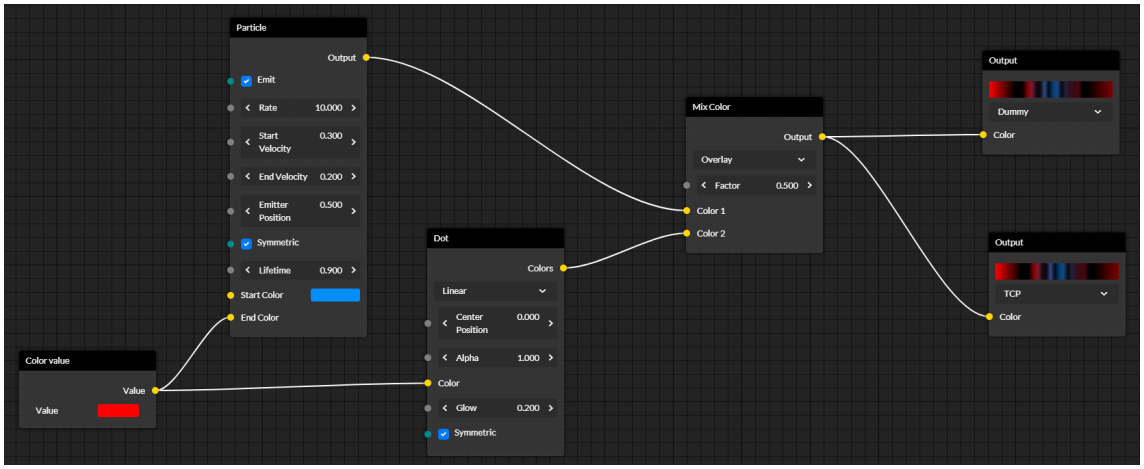

Figure 2.6: *baklava.js* example graph.

editor *baklava.js*[36] for *Vue* and *React Flow*[37] for *React*. Furthermore, there are different general purpose data visualization libraries, such as *D3*[38] and *Chart.js*[39]. For creating non-responsive UML diagrams there are libraries, such as *PlantUML*[40] or the discontinued *UmlCanvas*[41]. Figure 2.6 shows an example graph by *baklava.js*, where the individual attributes can be connected by edges.

2.3.5 Graph Layout Problems

Graph layout problems are a class of combinatorial optimization problems, where the goal is to find a linear layout of an input graph in such a way, that a particular objective function is optimized. Most interesting graph layout problems are NP-hard, although for most applications approximation algorithms are sufficient [33]. In this work, we make use of existing graph layout library *dagre*[42]. Its algorithm is based on the work from Barth et al. [9], Brandes and Köpf [15], Gansner et al. [41], Jünger and Mutzel [53], and Sander [85].

2.4 Ontologies

An ontology is an agreement among a broader community to commit to the same vocabulary. Once established, it enable combining and connecting data from different sources. In their paper ”What Is an Ontology?”, Guarino et al. [46] provide a definition for *computational ontologies*, as it is applied in the knowledge engineering community.

[36] https://github.com/newcat/baklavajs, accessed 2024/05/04.
[37] https://reactflow.dev, accessed 2024/05/04.
[38] https://d3js.org, accessed 2024/05/04.
[39] https://github.com/chartjs/Chart.js, accessed 2024/05/04.
[40] http://www.plantuml.com/plantuml/uml/, accessed 2024/05/05.
[41] https://github.com/christophevg/UmlCanvas, accessed 2024/05/05.
[42] https://github.com/dagrejs/dagre, accessed 2024/05/20.

> ”Computational ontologies are a means to formally model the structure of a system, i.e., the relevant entities and relations that emerge from its observation, and which are useful to our purposes. An example of such a system can be a company with all its employees and their interrelationships. The ontology engineer analyses relevant entities and organizes them into concepts and relations, being represented, respectively, by unary and binary predicates. The backbone of an ontology consists of a generalization/specialization hierarchy of concepts, i.e., a taxonomy. Supposing we are interested in aspects related to human resources, then Person, Manager, and Researcher might be relevant concepts, where the first is a superconcept of the latter two. Cooperates-with can be considered a relevant relation holding between persons. A concrete person working in a company would then be an instance of its corresponding concept.-[46]

In their article ”Knowledge Graphs”, Hogan et al. [51] give a comprehensive summary on how knowledge graphs are being used, what techniques are employed and how they relate to existing data management topics. They describe that representing data in a graph brings a number of benefits in settings that involve integrating, managing and extracting data from diverse sources at a large scale, compared to relational models. Most notably, it enables more flexibility in the data and allows postponing the schema definition, it supports navigational operators for recursively finding entities connected through arbitrary-length paths and it enables new ways of reasoning about the data, supporting machine learning techniques directly working with the graph. In the article, they outline graph data models (for example *RDF*), query languages (for example *SPARQL*), representations of schema (for example *RDF Schema (RDFS)* and *Shapes Constraint Language (SHACL)*), knowledge deduction (for example with *Web Ontology Language (OWL)*) and much more. Those are introduced in the following subsections.

2.4.1 Resource Description Framework (RDF)

RDF is a standard for establishing semantic interoperability on the Web. It provides a data model that can be extended to address sophisticated ontology representation techniques. The basic building block in RDF is an *object-attribute-value* triple [31], also referred to as a *subject-predicate-object* triple [51]. There are different notations for a triple, for example *Attribute(Object,Value)* and *(subject, predicate, object)*. See listing 2.3 for example data using those notations. A set of RDF triples represents a labeled, directed graph, with every triple representing an edge `[s]-p->[o]`. The subjects and objects of the triples are the nodes in the graph.

```
1 // A(O,V)
2 example.org:firstName('example.org/person42', 'Alex'}
3 example.org:lastName('example.org/person42', 'Smith'}
4
5 // (s, p, o)
6 ('example.org/person42', example.org/firstName, 'Alex)
7 ('example.org/person42', example.org/lastName, 'Smith)
```

Listing 2.3: Different notations for an RDF triple.

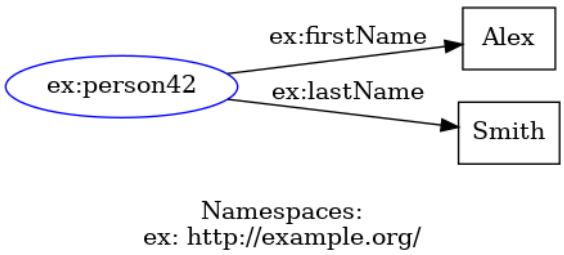

Figure 2.7: Visualization of the RDF graph defined in listing 2.4.

To unambiguously identify objects and vocabulary terms, a Uniform Resource Identifier (URI) [13] or Internationalized Resource Identifier (IRI) [35] is used. For every triple, the *subject* is an IRI or a blank node. The *predicate* is an IRI and the *object* is an IRI, a literal or a blank node. Blank nodes are nodes without an IRI, also called anonymous resources. To abbreviate IRI to qualified names, RDF syntaxes use namespaces [61]. Machine readable data formats for RDF are in XML [88], Turtle [11], JSON-LD [93], N-Triples [10], N-Quads [22] and others [44, 89]. Listing 2.4 shows our two example triples from listing 2.3 in Terse RDF Triple Language (Turtle) syntax, using namespaces. Figure 2.7 visualizes the RDF graph[43].

```
1 @prefix ex: <http://example.org/> .
2
3 ex:person42 ex:firstName "Alex" .
4 ex:person42 ex:lastName "Smith" .
```

Listing 2.4: RDF Graph in Turtle syntax.

RDF uses XML Schema datatypes for literal values, as shown in listing 2.5.

```
1 @prefix ex: <http://example.org/> .
2 @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
3
4 ex:person42 ex:birthDate "1996-05-26"^^xsd:date .
```

Listing 2.5: Usage of XML Schema datatypes for literals in RDF.

To define the *type* (also called *class* or *concept*) that an object is an instance of, the *type* predicate of the *rdf* namespace is used, as shown in listing 2.6. In the same manner, predicates and concepts from existing namespaces can be used, avoiding the need for repeating the work that others already have done and leading to shared vocabulary and interoperability.

For example, *Friend Of A Friend (FOAF)* [19] is an ontology for modeling social networks and already contains the predicates *firstName* and *lastName*, as well as the concept *Person*. Instead of defining our own predicates and concepts in the *https://example.org* namespace, we can use the ones from *FOAF*. Also, the `rdf:type` predicate can be abbreviated with "`a`". Furthermore, when defining

[43] The graph was visualized using `https://www.ldf.fi/service/rdf-grapher`, accessed 2024/09/27.

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

ex:person42 rdf:type ex:Person .
```

Listing 2.6: Type definition in RDF.

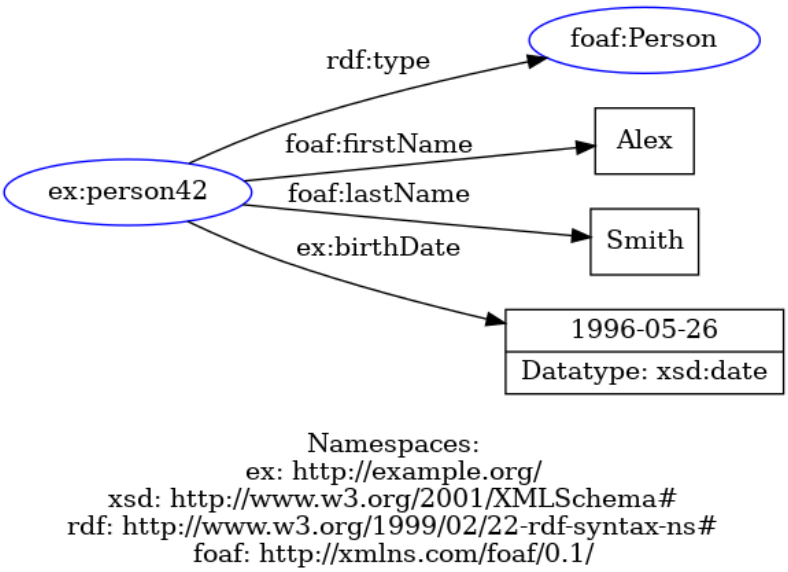

Figure 2.8: Visualization of the RDF graph defined in listing 2.7.

multiple predicates and objects for a subject, we can use a shortened syntax in Turtle. When we apply all of these points, we end up with the RDF graph definition shown in listing 2.7. Figure 2.8 shows the visualized RDF graph[44].

```
@prefix ex: <http://example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

ex:person42 a               foaf:Person ;
            foaf:firstName  "Alex" ;
            foaf:lastName   "Smith" ;
            ex:birthDate    "1996-05-26"^^xsd:date .
```

Listing 2.7: Complete RDF example.

[44] See footnote 43.

Table 2.2: Variable bindings for the query in listing 2.9 on the graph defined in listing 2.7.

?firstName	?lastName
“Alex‘	“Smith“

2.4.2 SPARQL Protocol and RDF Query Language (SPARQL)

SPARQL [48] is the standard language for querying RDF data [78]. Most forms of SPARQL query contain a set of triple patterns, which are like RDF triples, except that the subject, predicate and object may each be a variable. The syntax is Structured Query Language (SQL)-like, as shown in listing 2.8.

```
SELECT <variables>
FROM <graphs>
WHERE { <triple patterns> }
```

Listing 2.8: Syntax of a SPARQL query.

A query returns a set of variable bindings as result. For example, the query shown in listing 2.9 returns the variable bindings shown in table 2.2, given the RDF graph defined in listing 2.7. It binds every subject to the `?person` variable, for which there exists a triple with the `rdf:type` predicate and the `foaf:Person` object. Also, there must exist a triple with the same subject and a `foaf:firstName` predicate, as well as one with a `foaf:lastName` predicate. For all subjects, where this is the case, the variable bindings for `?firstName` and `?lastName` are returned.

```
SELECT ?firstName ?lastName
WHERE {
    ?person  rdf:type         foaf:Person .
    ?person  foaf:firstName   ?firstName .
    ?person  foaf:lastName    ?lastName .
}
```

Listing 2.9: Example SPARQL query.

SPARQL supports advanced features, such as *filters*, *group graph patterns*, *optional graph patterns*, *union graph patterns*, *ordering*, *limit*, *offset* and more. It supports other verbs besides *SELECT*, such as *CONSTRUCT* to dynamically construct a graph, or *ASK* to check whether the query pattern has a solution.

2.4.3 RDF Schema (RDFS)

RDFS [18] is a simple ontology and schema language for RDF. It is an RDF vocabulary and contains concepts and properties for defining concepts or properties and their characteristics. Listing 2.10 provides an example, where RDFS is used to make `foaf:Person` a subclass of `ex:LivingBeing` and to define the range and domain of the `ex:birthDate` property. Because of being a subclass of it, every `foaf:Person` inherits the properties of `ex:LivingBeing`. Furthermore, because of the range

and domain definition, for every triple that has `ex:birthDate` as predicate, the subject must be a living being (domain) and the object must be a date (range). Although they are defined as part of the RDF namespace, the terms `rdf:type` and `rdf:Property` are part of RDFS too. Analogous to `rdfs:subClassOf`, the hierarchy of properties can be specified using the term `rdfs:subPropertyOf`.

```
@prefix ex: <http://example.org/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

foaf:Person   rdfs:subClassOf   ex:LivingBeing .
ex:birthDate  rdfs:domain       ex:LivingBeing ;
              rdfs:range        xsd:date .
```

Listing 2.10: RDFS example.

2.4.4 Shapes Constraint Language (SHACL)

SHACL [57] is a schema language for RDF that is more expressive than RDFS. It validates an RDF graph against a set of so-called *shapes* and allows expressing complex constraints. Each shape declares targets (which nodes need to confirm to this shape), constraints and rules (what constraints do the target nodes have and which rules do they need to confirm to). Listing 2.11 shows an example shape that forces every node of type `foaf:Person` to have exactly one birth date as property, being of type `xsd:date`. Restricting the amount of properties that a node can or must have is a constraint that is not possible using only RDFS. SHACL focuses on data validation, based on the *closed-world assumption* (missing information is assumed to be false) and the *unique-name assumption* (distinct names refer to distinct objects).

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .

ex:PersonShape   rdf:type          sh:NodeShape ;
                 sh:targetClass    foaf:Person ;
                 sh:property [
                     sh:path      foaf:firstName ;
                     sh:minCount  1 ;
                     sh:maxCount  1 ;
                     sh:datatype  xsd:date ;
                 ] .
```

Listing 2.11: SHACL example shape.

2.4.5 Web Ontology Language (OWL)

OWL [70] is an ontology language for authoring ontologies. It allows expressing complex concept descriptions. Contrary to SHACL, the focus is less on data validation and more on inference (of new knowledge), based on the *open-world assumption* (missing information is assumed to be unknown) and *no unique-names assumption* (distinct names can refer to same object).

This work does not discuss SHACL or OWL any further and does not make use of them. They are introduced for the sake of completeness only.

2.4.6 JSON for Linked Data (JSON-LD)

JSON-LD [94] is a syntax to serialize graphs in JSON. It is fully compatible with JSON and introduces an *identifier mechanism* for JSON objects (via IRIs), a way for *disambiguating keys* shared among different JSON documents by mapping them to IRIs via a *context*, a mechanism in which a *value* of a JSON object *may refer to a resource* and more. The data model described by a JSON-LD document is an RDF graph. It can be combined with other RDF technology, such as SPARQL, but it can also be used without knowledge of RDF. Listing 2.12 shows the RDF graph from figure 2.8 in JSON-LD syntax.

```
{
  "@context": {
    "ex": "http://example.org/",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
    "foaf": "http://xmlns.com/foaf/0.1/",
    "ex:birthDate": {
      "@type": "xsd:date"
    }
  },
  "@id": "ex:person42",
  "@type": "foaf:Person",
  "foaf:firstName": "Alex",
  "foaf:lastName": "Smith",
  "ex:birthDate": "1996-05-26"
}
```

Listing 2.12: JSON-LD example object instances.

2.4.7 RDF and Ontology Visualization

There exist several techniques for visualizing an RDF graph or an ontology; see the work from Catenazzi et al. [23], Deligiannidis et al. [32], Frasincar et al. [39], Lohmann et al. [66], North et al. [74], and Silva et al. [91].

2.5 Bioinformatics Data Models

This master thesis is created in cooperation with the *Institute of Biochemistry and Technical Biochemistry* of University of Stuttgart. The intention is to apply *MetaConfigurator* for the creation and maintenance of data models in the field of biochemistry. This section introduces existing work on data models in the field.

EnzymeML [62, 81, 82] is a data model for biocatalysis and enzymology. *Strenda DB* [97] is a data model for enzyme kinetics data. For both of these data models, there exists a JSON schema. Software-driven RDM (sdRDM)[45] is a research project at University of Stuttgart to "*develop and implement an RDM concept which integrates software and data formats based on an object-based data model, and to apply it to the analysis of enzymatic data*". Within the scope of that project, Jan Range et al, developed a schema language based on the Markdown [45] data format[46] and a web based markdown schema editor.

NFDI4Chem [95] is a national initiative to standardize chemical research data infrastructure in Germany, in agreement with the *Findable, Accessible, Interoperable, and Re-usable (FAIR)* [102] principles. One of their objectives is to initiate international community processes to establish minimum information standards for data and machine-readable metadata, to identify and recommend open data standards in key areas of chemistry and develop standards if there is a lack. Also, they want to promote the use of digital tools in all stages of research and promote subsequent research data management at all levels in academia. Furthermore, the *Allotrope Foundation*[47], an international consortium of pharmaceutical, biopharmaceutical and other industries, aims to develop a standard and architecture for collection, exchange and management of laboratory data. They consider the approaches *Standardization in Lab Automation (SiLA)* [8] or *Open Platform Communication - Unified Architecture (OPC UA)* [65] as approaches for the exchange of information between different platforms or providers [42]. On the other hand, there already exist several ontologies in the field of chemistry and biology, such as *National Cancer Institute Thesaurus* [43], *Chemical Methods Ontology*[48] and others [96].

[45] https://www.simtech.uni-stuttgart.de/exc/research/pn/pn2/pn-2-6-ii/, accessed 2024/09/24.
[46] https://github.com/FAIRChemistry/software-driven-rdm, accessed 2024/09/24.
[47] https://www.allotrope.org, accessed 2024/09/26.
[48] https://github.com/rsc-ontologies/rsc-cmo, accessed 2024/09/26.

3 Design and Implementation

This chapter is the core of the thesis. It describes the design and implementation of new functionalities for *MetaConfigurator*.

3.1 Architecture Rework

The architecture of *MetaConfigurator* offers enough modularity to support different data formats and implement different *panels*, such as the *text editor panel* or the *GUI editor panel*. However, it is also lacking in multiple aspects. In this section, those limitations are addressed, and architectural improvements are designed and implemented. Also, from now on this paper refers to the *File editor* tab/mode as *Data editor*. Within the source code it was renamed too.

3.1.1 Managing Multiple Schemas and Data Files Simultaneously

The original design features a single source of truth store, which is responsible for the currently loaded schema and configuration file. There can only be one schema and configuration file loaded at any given time. When the user changes the tool *mode* $\in$ {*Data editor*, *Schema editor*, *Settings*} (tab), the schema and configuration file in the store are replaced accordingly. Table 2.1 illustrates which data and schema are used for the different modes. Most subcomponents of the tool access the central store directly, instead of receiving the current data and schema as a parameter passed through the component hierarchy. This comes with one major drawback: it is not possible to have the schema and data of different views loaded at the same time. However, being able to simultaneously show content of different modes could significantly help the user. For example, in the schema editor view, the data editor GUI could be shown too, giving the user a preview of the GUI that their schema will result in.

To get rid of this limitation, the $sessionStore = \{currentSchema, currentData, ...\}$ is removed. Instead, a new store is created, which for each *mode* stores the corresponding *schema*, *data*, *user selections* (e.g., which path is currently selected and which properties in the tree are expanded) and *validation results*. Additionally, mapping functions (e.g., $getDataForMode : mode \rightarrow data$) are introduced, with which the content can be accessed. Instead of the subcomponents of *MetaConfigurator* always accessing the one current data and schema, they now each receive a *mode* in their constructor and access the corresponding data. For example, we can instantiate a text editor panel with the mode *data editor*. This panel will now use data and schema of the *data editor* mode. This architecture allows us to arrange panels of different types and different modes in an arbitrary manner, which brings us to our next point.

Table 3.1: Panels and their data sources for the different modes.

Tool Mode	Panel 1: type(mode)	Panel 2: type(mode)
Data editor	Text editor(data editor)	GUI editor(data editor)
Schema editor	Text editor(schema editor)	GUI editor(schema editor)
Settings	Text editor(settings)	GUI editor(settings)

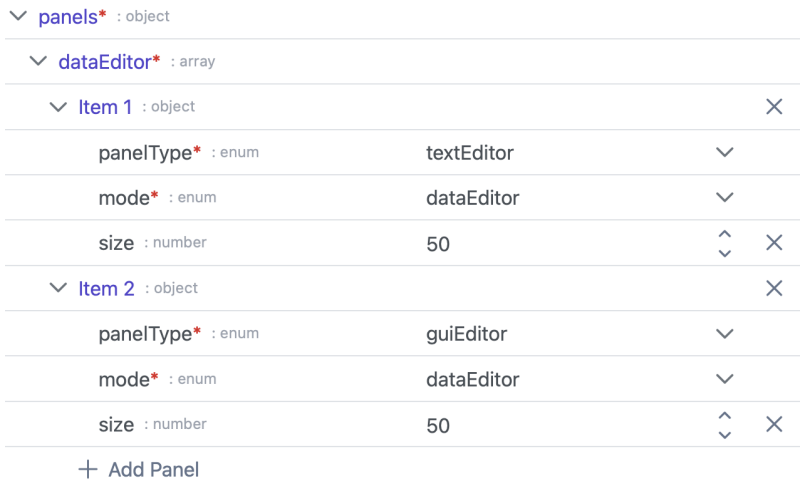

Figure 3.1: Settings GUI editor view with panels configuration for *data editor* mode.

Table 3.2: Modified schema editor panel configuration.

Tool Mode	Panel 1: type(mode)	Panel 2: type(mode)	Panel 3: type(mode)
Schema editor	Text editor(schema editor)	GUI editor(schema editor)	GUI editor(data editor)

3.1.2 Dynamic Panel Arrangement

The *text editor* or *GUI editor* are what we call *panels*. They have access to the *data* and *schema* and visualize them in a particular manner. They can also update the *data* or the *currently selected element*, which are synchronized across all panels. *MetaConfigurator* can be extended by more panels. Table 3.1 shows which panels are used for the different tool modes, and their assigned panel modes (which data and schema to use). For every mode, *MetaConfigurator* provides a text editor panel and a GUI editor panel, both using the data and schema of the corresponding mode.

To allow for more variation in the panels configuration (arrangement), we turn it into a setting. This setting can be adjusted by the user (see figure 3.1) or by *MetaConfigurator* itself. Table 3.2 shows an alternative configuration for the panels of the schema editor. It contains an additional GUI editor panel, which uses the data and schema from data editor mode. This configuration results in a schema editor, where, on top of the regular text editor and GUI editor, the user gets a (functional) preview of the GUI that their schema will generate, see figure 3.2.

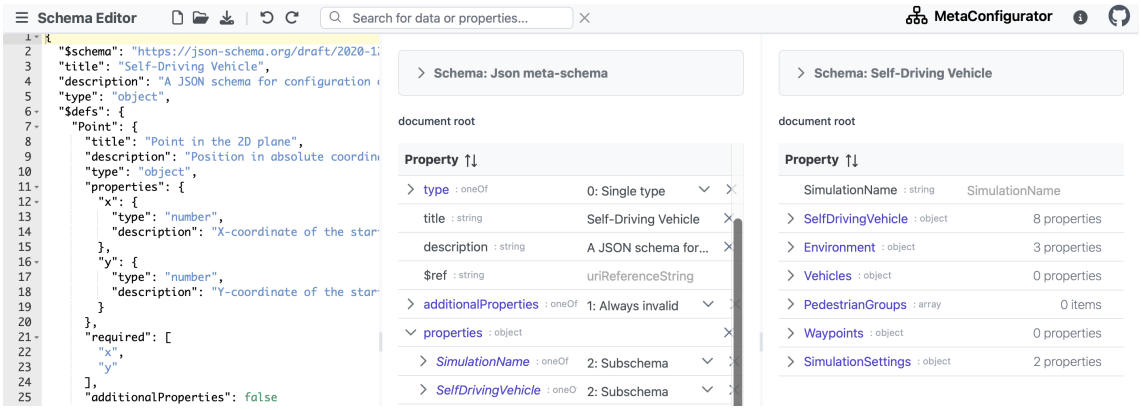

Figure 3.2: Schema editor with a preview of the resulting GUI on the right.

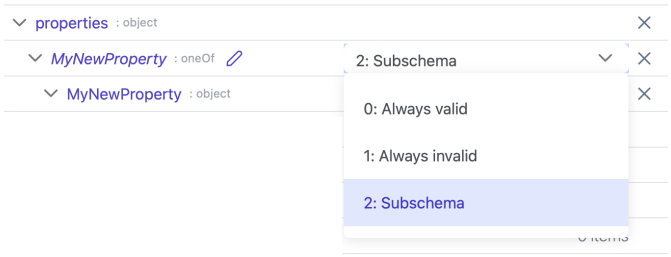

Figure 3.3: Drop-down menu when defining a new property: selection of either a boolean schema or a sub-schema.

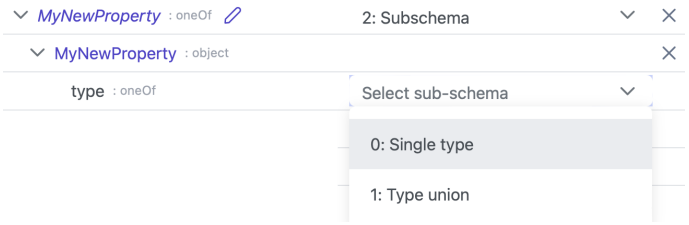

Figure 3.4: Drop-down menu when defining a new sub-schema: selection of either a single type or a type union.

3.2 Simplified and Customizable Schema Editor

The *MetaConfigurator schema editor* can be complicated for the user, especially if they have never worked with JSON schema before [72]. This section describes how the *MetaConfigurator* schema editor is made easier to use for the user.

3.2.1 Making the Schema Editor More Simple

The reason for the complexity of the schema editor lies in the expressiveness of JSON schema and because *MetaConfigurator* supports most of its keywords and features. This is illustrated by figures 3.3-3.5: when creating a new property for an object, JSON schema allows for the property to be a *boolean schema* (`true` or `false`) or a *sub-schema*. For every *sub-schema*, the `type` can either be empty, an array of types or a singular type. All those options that JSON schema offers result in options that the user of the schema editor has to navigate through.

Other schema editors, such as Adamant [90] are easier to use, because they are simpler and do not support as many keywords and options as *MetaConfigurator*. In practice, many JSON schema keywords are used infrequently [4] and the full expressiveness of JSON schema is not needed. To allow for an easier schema editing experience, while maintaining the expressiveness and support for most JSON schema keywords, we introduce JSON meta schema parameters. The user can decide themselves which advanced JSON meta schema features they want to use, and where they want to work with the simplified meta schema. The following boolean parameters are introduced:

- **allowBooleanSchema:** Whether a JSON Schema definition can also be just `true` or `false`. Having this option enabled will increase the choices that have to be made when defining a sub-schema in the schema editor.

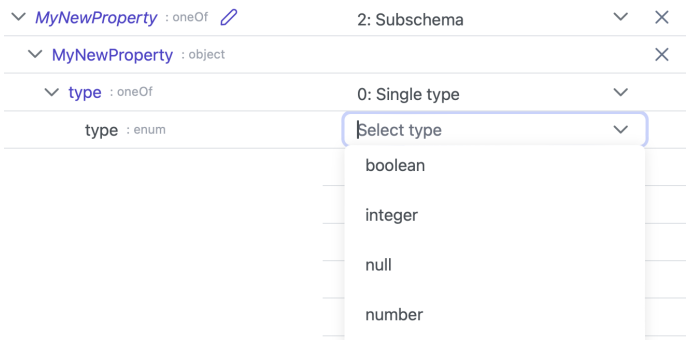

Figure 3.5: Drop-down menu when defining a single type.

- **allowMultipleTypes:** Whether an object property can be assigned to multiple types (e.g., `string` and `number`). Having this option enabled will increase the choices that have to be made when defining the type of a sub-schema in the schema editor, but it also allows more flexibility. An alternative to defining multiple types directly is using the `anyOf` or `oneOf` keywords.
- **showAdditionalPropertiesButton:** The meta schema allows additional properties in the schema that are unknown to the meta schema. To support this in the schema editor, it would always provide an *Add Property* button, not only for specifying object properties but also to specify additional properties for numbers, strings and booleans. In practice, this option is not used much but it can confuse the user. When the setting is disabled the button is not shown to the user.
- **objectTypesComfort:** This is a comfort feature: the original JSON meta schema allows properties of any particular type to have example values, constant values, default values or enum values of different types. For example, a field for numbers could have a string as a default value. This meta schema option forces the same type for all these fields. This avoids the need for the user to manually select the types.

  Warning: due to incompatibility, this option will disable schema editor support for defining the items of an array, as well as support for many advanced keywords, such as conditionals and `not`.
- **markMoreFieldsAsAdvanced:** Some advanced features, such as conditionals (e.g., `if`) and composition (e.g., `oneOf`), are located within an *advanced* section within the GUI and are shown only when this section is actively expanded by the user. This avoids overwhelming them with options. Enabling this setting will move also slightly more simple properties to the advanced section, such as `default`, `examples`, `enum`, `const` and `string length`.

The customizability of the schema editor is achieved by a novel approach, which we call *meta schema builder*. Based on the meta schema parameters, it builds a custom JSON meta schema, which is then used to create the schema editor GUI.

A toolbar button is added to the schema editor (figure 3.6), which allows the user to toggle between an advanced meta schema (boolean schema and multiple types allowed, no object types comfort feature and showing the button to add additional properties) and a simple one (opposite configuration). Furthermore, in the settings menu, the user can set the values for the individual parameters. Figure

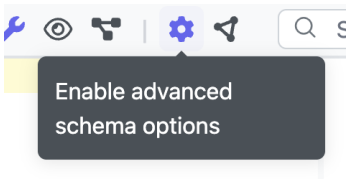

Figure 3.6: Button to enable the advanced schema editor, by using a more advanced meta schema.

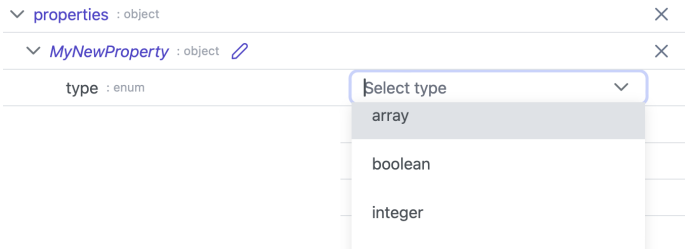

Figure 3.7: Drop-down menu when defining the type of a new property.

3.7 shows the drop-down menu for defining the type of a property, when using the simplified meta schema. The type can be set directly. Compare this with the advanced meta schema, where first the user had to select `sub-schema`, then `single type` and only then could select the actual type.

3.2.2 Providing a (Functional) Preview of the Resulting GUI

The data editor GUI, which is created from the user schema, helps the user understand their schema. It shows the properties defined in the schema, the titles and descriptions in the schema, whether a property is `required` or `deprecated`, and more. To access the data editor, the user has to navigate from the schema editor view to the data editor view. They cannot view both the schema editor and the resulting GUI simultaneously. However, the improved architecture, as described in section 3.1.2, does make different panel arrangements possible. We make use of that architecture and add a button to the schema editor (figure 3.8), which allows the user to toggle between showing the resulting data editor GUI in a third panel and hiding it. Another improvement for the schema editor is the interactive schema diagram, which is described in section 3.6.

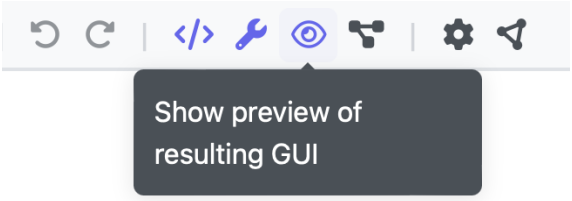

Figure 3.8: Button to show the data editor GUI in a third panel.

Field	Title of the work	Authors	Year
Research	Chemometrics: data analysis for the laboratory and chemical plant	Brereton [17]	2003
Research	Practical data analysis in chemistry	Maeder and Neuhold [67]	2007
Research	Meta-analyses and Forest plots using a microsoft excel spreadsheet: step-by-step guide focusing on descriptive data analysis	Neyeloff et al. [73]	2012
Research	Excel data analysis	Guerrero et al. [47]	2019
Research	Using Excel and Word to structure qualitative data	Ose [76]	2016
Research	Data organization in spreadsheets	Broman and Woo [20]	2018
Research	Using Microsoft Excel to code and thematically analyze qualitative data: a simple, cost-effective approach.	Bree and Gallagher [16]	2016
Industry	The popularity of data analysis software	Muenchen [71]	2012
Industry	Excel-lence in data visualization?: the use of Microsoft Excel for data visualization and the analysis of big data	Raubenheimer [83]	2017
Industry	Implementation of the LPS using an excel spreadsheet: A case study from the New Zealand construction industry	Zaeri et al. [104]	2017
Industry	Excel-based application of data visualization techniques for process monitoring in the forest products industry.	Cook et al. [29]	2004
Industry	Advanced analysis of manufacturing data in Excel and its Add-ins	Kajáti et al. [54]	2017

Table 3.3: Usage of Excel in Research and Industry.

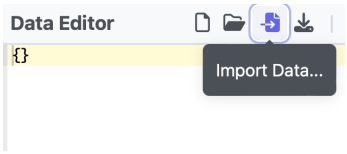

Figure 3.9: Menu Option for importing data.

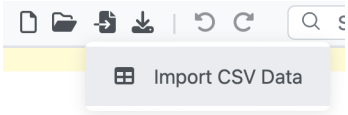

Figure 3.10: Menu Option for importing CSV data.

3.3 CSV Import

Microsoft Excel is a popular [71, 83] tool for managing data in tables, used in research and industry alike, as illustrated by table 3.3. While being easy to read, understand and modify by humans, Excel spreadsheets lack a clear schema. This makes them suboptimal for automated processing by machines. JSON, on the other hand, is designed to be machine-readable and through JSON schema, a particular data structure can be enforced.

To make it easy for the user to port data from an existing Excel spreadsheet (or any CSV file) into a JSON document, an import functionality is added to *MetaConfigurator*. Upon clicking the corresponding button in the toolbar (figure 3.9 and figure 3.10), an import dialog is shown to the user, where they can select and upload their local CSV document. Once a document is uploaded, *MetaConfigurator* analyses the content to predict the delimiter and decimal separator (based on the frequency and position of the characters in the document). The predicted delimiter and decimal separator characters are pre-selected but could be modified by the user. Now the user has two options: importing an independent table, or expanding an already imported table with a second table.

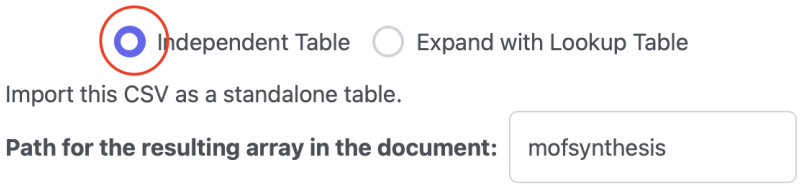

Figure 3.11: Importing an independent table.

3.3.1 Option A: Importing an independent table

A CSV document can be represented as an array of objects in JSON, each object sharing the same schema. When importing an independent table into *MetaConfigurator*, the user has to specify the name of the path, where the array should be imported into (figure 3.11). By default, this path is the name of the Excel document, without special characters and in lower case. However, more complex paths, with hierarchies and array indices, are also supported, allowing the user to import the data to any position of their choice within the current document. The document can be an empty and new document but it can also be a large document where already many properties exist and, possibly, more tables were imported before. For example, the EnzymeML schema (see section 2.5) has an array of *creators*, an array of *vessels*, an array of *proteins* and more. The data of each of these arrays could be imported from separate CSV files.

3.3.2 Option B: Expanding with Lookup Table

Alternatively, the new table can be used to expand an attribute of a previously imported document, similar to a `JOIN` operation in SQL, see figure 3.12. The user specifies which property in the new CSV is the primary key (A). Also, they specify which existing table to expand (B; *MetaConfigurator* automatically scans the document for arrays of objects and suggests options). Finally, they specify which property of the existing table acts as a foreign key, linking the first table with the second table (C). *MetaConfigurator* automatically iterates over all properties of the first table and computes for which property there are most matches of primary and secondary keys. The property with the most matches is automatically selected, but the selection can be changed by the user too. When pressing `import`, *MetaConfigurator* iterates over every object in the array from the first table and then tries to match the foreign key of that object with the primary key of the table to import. If a match is found, the primitive foreign key attribute (usually a string) is replaced by the object from the new CSV. Example: first, the user imports an *experiments* table, with each experiment having a `compound` property of the type string (compound identifier). Next, the user uploads a *compounds* table, with each entry describing the properties (molecular weight, smiles code) of a compound. The experiments table has the `compound` foreign key and the compounds table has `compound` as primary key. Instead of importing the *compound* table independently, the user can expand the `compound` string property of the already imported experiments with the *compound* lookup table. This will turn the `compound` string property into an object property, with the molecular weight and smiles code added as child properties and the values being taken from the compounds table. For more information see section 4.3 in the application chapter.

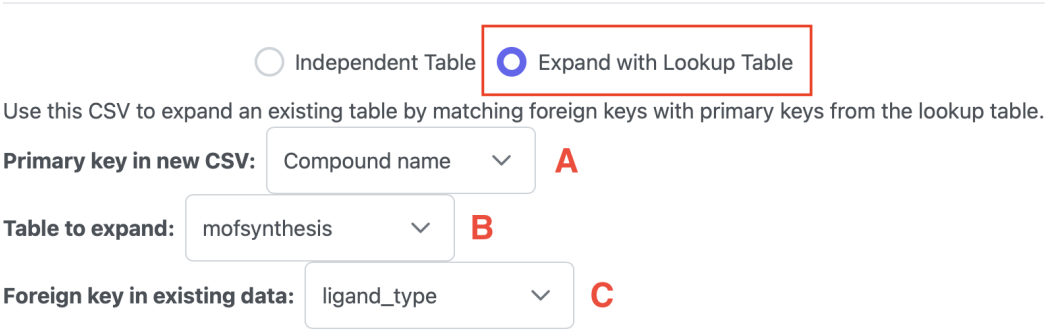

Figure 3.12: Expanding with Lookup Table.

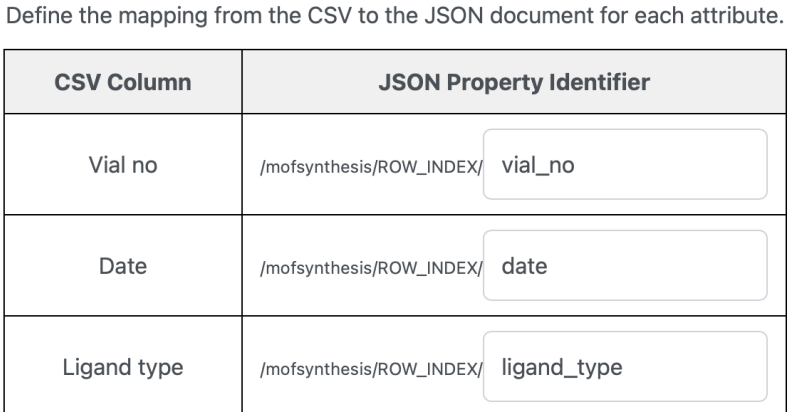

Figure 3.13: Mapping from CSV to JSON.

3.3.3 Defining the Mapping from CSV to JSON

In both options 1 and 2, the user can specify the mapping from the columns in CSV to the names and paths of the properties in JSON (see figure 3.13). By default, *MetaConfigurator* suggests the column name transformed to snake case as path for the properties. More complex paths and nesting are supported.

3.3.4 Schema Inference

While a JSON schema can be crafted by hand, there also exists literature on inferring a schema from existing data by Baazizi et al. [2, 3, 5] and Čontoš and Svoboda [28]. Such methods might not be perfect. For example, they might infer that a particular field is of type `string` but do not recognize it as an enumeration, although in practice it is. Also, the approaches tend to not model cyclical dependencies but instead generate a deeply nested schema. Yet, they can save the user a lot of work and generate an initial schema document, which the user could then further adjust. Schema inference functionality was added to *MetaConfigurator* using the *jsonhero/schema-infer*[1]

[1] https://github.com/triggerdotdev/schema-infer accessed 2024/06/04

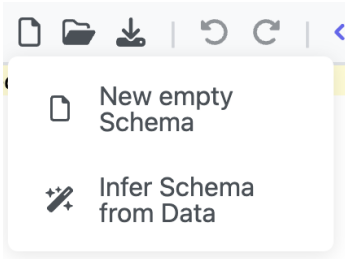

Figure 3.14: Menu Option to Infer Schema from Data.

library. In the *CSV Import* dialog, the user can specify whether the corresponding schema should be automatically inferred or not. Also, outside of *CSV Import*, a button for schema inference (based on the data in the *data editor*) was added to *MetaConfigurator*, see figure 3.14. Listing 3.1 shows a sample JSON document and listing 3.2 the inferred schema from the document.

```json
{
  "numberProp": 5,
  "stringProp": "hello",
  "booleanArrayProp": [
    true,
    false
  ],
  "objectArrayProp": [
    {
      "firstName": "Alex",
      "lastName": "Smith",
      "age": 28
    },
    {
      "firstName": "John",
      "lastName": "Doe"
    }
  ]
}
```

Listing 3.1: JSON example for inference.

3.3.5 Limitations

The CSV import does currently not support importing multiple documents into the same JSON path. Instead of appending the data, it is replaced. In the future it could be an option to append new data instead.

```
{
  "type": "object",
  "properties": {
    "numberProp": {
      "type": "integer"
    },
    "stringProp": {
      "type": "string"
    },
    "booleanArrayProp": {
      "type": "array",
      "items": {
        "type": "boolean"
      }
    },
    "objectArrayProp": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "firstName": {
            "type": "string"
          },
          "lastName": {
            "type": "string"
          },
          "age": {
            "type": "integer"
          }
        },
        "required": [
          "firstName",
          "lastName"
        ]
      }
    }
  },
  "required": [
    "numberProp",
    "stringProp",
    "booleanArrayProp",
    "objectArrayProp"
  ]
}
```

Listing 3.2: Inferred JSON schema based on data from listing 3.1.

3.4 Sharing Snapshots

To increase collaboration and make it easier for users to share their data and schemas with others, two functionalities are added to *MetaConfigurator*: First, support for query string parameters is added. When appending the path to a data, schema or settings file to the tool web address, *MetaConfigurator* attempts to load these files on startup. Links with these query string parameters can be shared with others, who immediately get the corresponding content loaded when accessing the link. Second, a backend that can persist user data is introduced.

3.4.1 Query String Support in Frontend

We implement support for loading pre-defined data/schema/settings based on the *query string* in the Uniform Resource Locator (URL), in the frontend. This is done by introducing the following optional query string parameters:

- **data:** URL of the data file to load.
- **schema:** URL of the schema file to load.
- **settings:** URL of the settings file to load.

While *MetaConfigurator* normally expects a JSON document when accessing the URLs from a query string, special support for GitHub URLs is added. Every URL to a GitHub file (not raw content but Hypertext Markup Language (HTML) document) is converted to a URL that leads to the raw file content. This way, users can link their GitHub files without having to worry about whether they provide the HTML link or raw file. When loading a schema via the query string, the initial schema selection dialog is not shown to the user.

Additionally, we introduce three new settings properties to customize the UI of *MetaConfigurator*. This enables users to share links to the tool with a customized look:

- **toolbarTitle**: defines the title that is displayed on the *MetaConfigurator* toolbar on top of the page.
- **hideSchemaEditor**: if set to `true`, the complete schema editor tab will be hidden.
- **hideSettings**: if set to `true`, the complete settings editor tab will be hidden.

Figure 3.15 shows the resulting view of *MetaConfigurator*, when opening it using the exemplary URL domain.org/meta-configurator/?data=a&schema=b&settings=c, with *a*, *b*, *c* being URLs to a data, a schema and a settings file. *MetaConfigurator* is opened in the *data editor* mode, with a pre-defined *Self-Driving Vehicle* schema, data file and settings. The settings file defines the custom title ”Autonomous Vehicle Editor”, which we can see in the top toolbar in the figure. Notice that the user will not see the usual schema selection dialog and will not have empty files, but instead start with the pre-defined data. This functionality does **not** rely on the backend to work. Furthermore, upwards-compatibility of data and schemas with new *MetaConfigurator* versions is guaranteed, because the schema follows the JSON schema standard, which is not *MetaConfigurator*-specific, and because the data follows the constraints of the schema, which also is tool-independent. To ensure compatibility of the settings with new *MetaConfigurator* versions, several mechanisms are introduced: 1. for every setting not present in the file, but expected by *MetaConfigurator*, the default

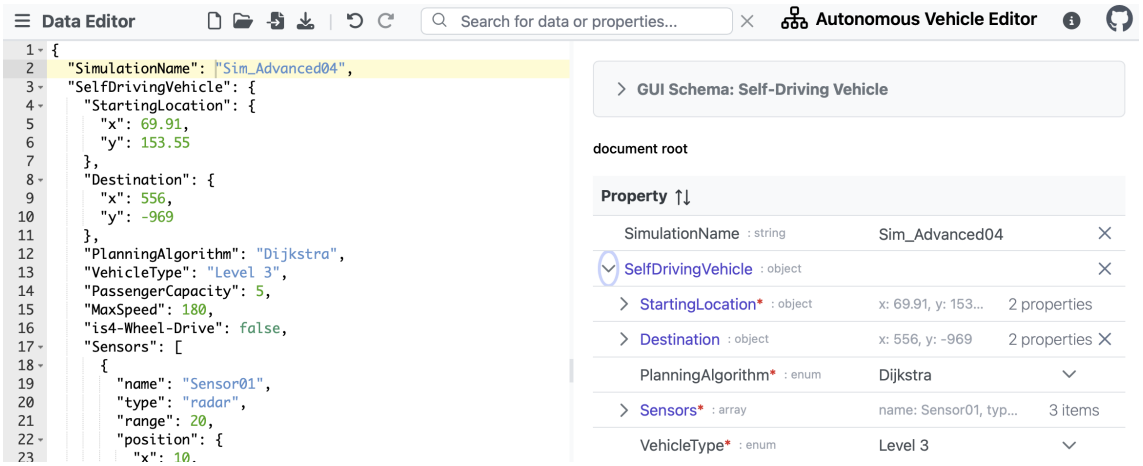

Figure 3.15: *MetaConfigurator* with a pre-defined *data*, *schema* and *settings* file. The settings file has the property *toolbarTitle* set as "Autonomous Vehicle Editor".

value for that setting is appended to the settings document, 2. if the active panels should be unknown to the tool (e.g., due to renaming), the *text panel* will be used by default, 3. a *restore default settings* button is added to the settings screen.

3.4.2 Storing Snapshots in the Backend

To enable persisting data for the user, a backend is introduced. The primary aim is to persist tuples of the user *data*, *schema* and *settings*. We call each such tuple a *snapshot*. After creation, snapshots can be restored via backend at a later time. This serves the following purposes:

- Users can save a snapshot of their session and load it at a later time to continue their work.
- Users can generate a snapshot of their session to share it with other people (or devices), who can work on it too.
- Users can define their own schema and then create a snapshot. By loading the snapshot, other people will get a user-friendly form to edit data according to this schema. As a result, everyone can create and share their own configurator.

A backend service is set up using the Python *Flask*[2] library. It is connected with a *MongoDB*[3] database, where it stores snapshots of the users. The backend offers a Representational State Transfer (REST) Application Programming Interface (API) to store and retrieve snapshots, supporting the following kinds of requests:

[2] https://flask.palletsprojects.com/en/3.0.x/, accessed 2024/08/19.
[3] https://www.mongodb.com, accessed 2024/08/19.

POST */snapshot*

Expects a tuple of *data*, *schema* and *settings*, optionally a *snapshot_id*. The request will fail if the files exceed a certain size limit or the ID is already in use. If no ID is given, it will create a new random Universally Unique Identifier (UUID). Stores the snapshot in the MongoDB under the ID and returns the ID to the user. Also stores *creation date*, *last access date* and *access count* for the snapshot in the MongoDB.

GET */snapshot/<snapshot_id>*

Returns the snapshot with the given ID, if existing. Updates the *last access date* and *access count* of the snapshot.

POST */project*

Expects the same data as **POST */snapshot***, plus a *project_id* and *edit_password*. Fails if the *project_id* is already in use or the *edit_password* is deemed unsafe. If the *project_id* is already in use and the correct *edit_password* is given, then the existing project is updated instead. Stores the snapshot in the same manner as for the **POST */snapshot*** request, but also adds a new *project* entry in the database, storing a hash of the *edit_password* and a reference to the *snapshot_id*. Updating an existing project will result in updating the *snapshot_id*.

GET */project/<project_id>*

Same as for **GET */snapshot/<id>***, except it uses the *project_id* to determine the corresponding *snapshot_id*.

3.4.3 Periodic Backend Cleanup

To ensure the database does not get filled up with unused data, every 24 hours, the backend is cleaned up: Projects that have not been accessed for 90 days are deleted. Snapshots that are neither referred to by a project, nor accessed since 30 day, are deleted. Note that the clean-up mechanism and specifically the time thresholds are up for debate and might be modified in the future. Furthermore, to avoid abuse of the backend API, rate limits are defined. Only two projects or snapshots can be posted per minute, and only 20 snapshots or 10 projects can be received per minute. Rate limiting is implemented using the *Flask-Limiter*[4] library.

[4] https://flask-limiter.readthedocs.io/en/stable/, accessed 2024/08/19.

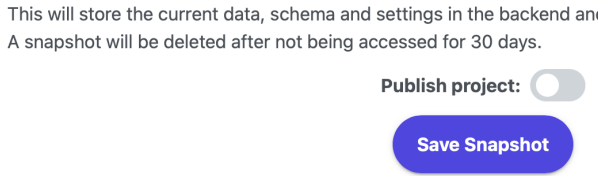

Figure 3.16: Dialog for creating a Snapshot.

3.4.4 Docker Compose Setup

Docker Compose[5] is used for deploying the backend, frontend and MongoDB in an automated manner. The backend and frontend are built using a *Dockerfile*[6] for each, with instructions for the build process. The built frontend files are copied and hosted by a *nginx*[7] web server and reverse proxy. For the backend *gunicorn*[8] is used for running it in production. To distribute rate limiting information across multiple instances of the backend, *Redis*[9] is used. This makes the backend scalable. Redis supports in-memory data storage and can also persist the rate limiting information on the disk.

3.4.5 Creating Snapshots and Projects in the Frontend

A button is added to the *MetaConfigurator* UI to open the snapshot creation dialog. The user can choose between creating a snapshot or creating a project. Figure 3.16 shows the dialog for creating a snapshot. When the user presses the *Save Snapshot* button, schema, data and settings are sent to the backend. The backend stores them and returns a `snapshot_id`. The URL to open *MetaConfigurator* with the snapshot loaded is displayed to the user. Figure 3.17 shows the dialog for saving a project. The user has to enter a project ID and edit password. If the project ID already exists or the password is not deemed safe, then an error message is shown to the user (figure 3.18). Note that the button for snapshot creation is currently located in the settings tab only. This is because the final stable hosting of the backend has not yet been achieved. The backend software is working, stable and scalable, but due to restrictions on the university server, it might have to be moved to a different server. Once a high uptime of the backend can be guaranteed, the snapshot sharing feature will be shown to the user in a more prominent way.

[5] https://docs.docker.com/compose/, accessed 2024/08/19.
[6] https://docs.docker.com/reference/dockerfile/, accessed 2024/08/19.
[7] https://nginx.org/en/, accessed 2024/08/19.
[8] https://gunicorn.org, accessed 2024/08/19.
[9] https://redis.io, accessed 2024/08/19.

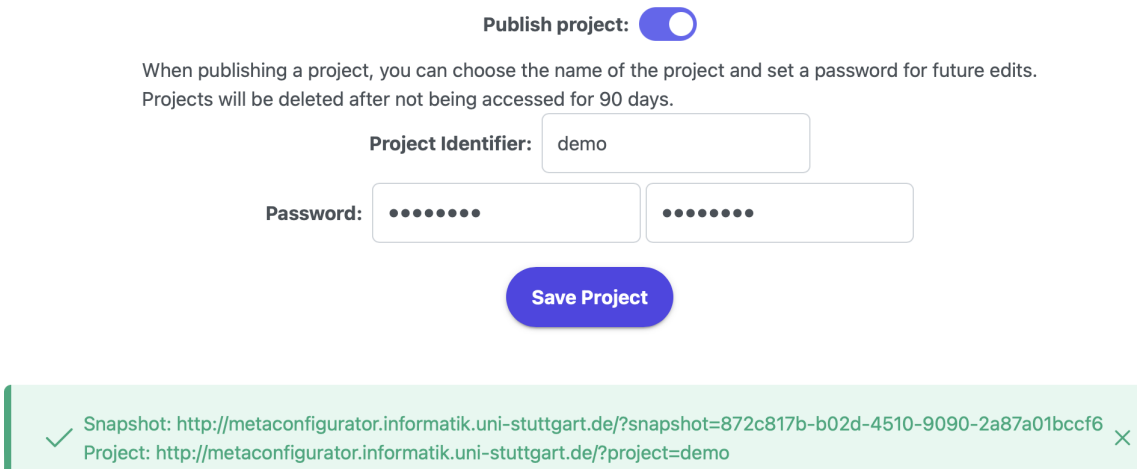

Figure 3.17: Dialog for creating a Project, with result already shown.

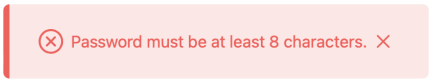

Figure 3.18: Error message when attempting to create project with short password.

3.5 Ontology Integration

Already, without special tooling, JSON documents can be organized in such a way that they represent an RDF graph (JSON-LD syntax). To provide additional assistance to the user, a new setting for the *meta schema builder* is introduced, which optionally provides JSON-LD fields in the schema editor. If enabled, it shows a `@context` property on schema root level, where the context can be defined (e.g., prefixes), an `@id` property for linking a JSON schema object to an RDF IRI, and a `type` property to define the RDF type of a JSON schema object. The context is a dictionary of key-value pairs, with the values being either directly an *IRI* or an object with the (optional) properties `@id`, `@type`, `@value` (advanced) and `@container` (advanced). Furthermore, for every JSON object schema, it shows the (optional) properties `@id`, `@type`, `@value` (advanced) and `@container` (advanced). Figure 3.19 shows an exemplary context defined for a schema.

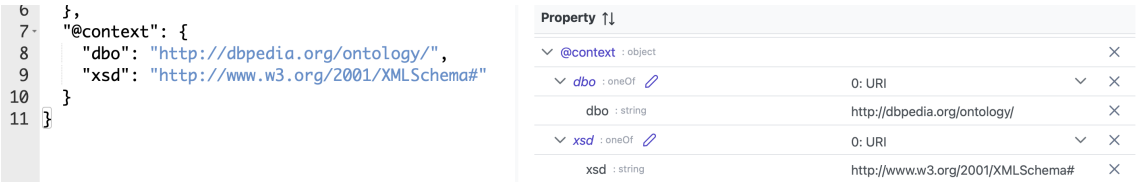

Figure 3.19: Example of a JSON-LD context defined in *MetaConfigurator*.

3.5.1 IRI Auto-Completion

To assist the user, auto-completion of ontology IRIs is implemented. Instead of showing a regular text field (as done for other strings) for IRIs, the *GUI editor* panel shows two *AutoComplete* fields: one short field for the optional prefix and one longer field for the rest of the IRI. The field for the prefix allows any string but makes suggestions based on the prefixes defined by the user in the `@context`. The IRI field also allows any string but makes suggestions based on the user input. *MetaConfigurator* sends a SPARQL query to the SPARQL endpoint defined in the user's settings, querying for all RDF classes and properties which contain the user input (search term) in their name, ordered by similarity. The results are returned to the user, allowing them to select from existing IRIs. This way, users of *MetaConfigurator* can create their own custom JSON schemas and map them to existing ontologies, making their data more connected, standardized and increasing interoperability. Figure 3.20 shows the auto-completion suggestions provided by *MetaConfigurator*, based on the user input `person`. The IRIs are retrieved by querying the *DBpedia SPARQL endpoint*[10], which is defined as the default SPARQL endpoint in the settings and can be changed to any other SPARQL endpoint. Figure 3.21 shows the same form, now with a prefix selected from the given context (see figure 3.19). The suggested IRIs are already filtered by prefix. Hence, only the IRIs starting with `http://dbpedia.org/ontology/` are shown in figure 3.21, but not the ones starting with `http://dbpedia.org/property/`.

[10] https://www.dbpedia.org/resources/sparql/, accessed 2024/08/19.

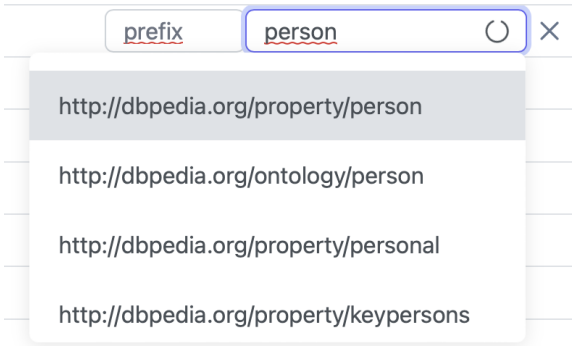

Figure 3.20: Example of auto-completion for IRIs using the given SPARQL endpoint with no prefix defined.

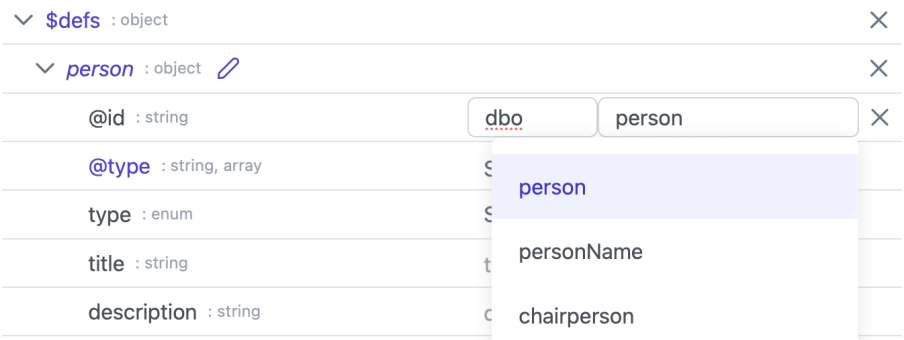

Figure 3.21: Example of auto-completion for IRIs using the given SPARQL endpoint with dbo as prefix.

3.5.2 Supported JSON-LD Fields

The following JSON-LD fields are implemented in *MetaConfigurator* for the *GUI panel* in the schema editor:

`@context`

Explained above.

`@id`

To map a JSON object to an RDF IRI, see above.

`@type`

Equivalent to the `rdf:type` predicate. For object schemas, `owl:Class` could be used as a type (with `owl` being the prefix for `http://www.w3.org/2002/07/owl#`). For the schema of an object property, the type `owl:ObjectProperty` could be used.

When using JSON-LD on the instance level (*data editor* instead of *schema editor*), then types from the schema would be used. For example, if we create instances of a `person` (e.g., Alice and Bob) and the `person` schema is mapped to the `http://dbpedia.org/ontology/person` class, then the `@type` of the instances would be the `http://dbpedia.org/ontology/person` too.

`@value`

This property is used to represent a primitive value (string, number, boolean). The reason this is an option, instead of just writing down the value directly, is because in some scenarios one might want to annotate more information to the value. Listing 3.3 shows a JSON-LD document defining the name of a `person` named *Karl*. The same document is shown in listing 3.4, now with the additional metadata that the name is in German language.

```
{
    "name": "Karl"
}
```

Listing 3.3: Example JSON-LD document defining a value without metadata.

```
{
  "name": {
     "@value": "Karl",
     "@language": "de",
  }
}
```

Listing 3.4: Example JSON-LD document defining a value without metadata.

Note that any other JSON-LD keyword can be added to schemas using the *text panel* instead of the *GUI*. To show JSON-LD fields in the *GUI* on instance level, in the *data editor*, they have to be added as properties to the schema. This is because JSON-LD can be used in different ways and it supports many possible keywords (e.g., `@included`, `@graph`, `@nest`, `@reverse`, etc.). Showing every possible keyword to the user for every single JSON object would overwhelm them. Instead, it is the task of the schema maintainer to define which properties their data models should have.

On instance level, IRI auto-completion is achieved by adding `metaConfigurator/ontology/mustBeUri: true` metadata to a string property in the schema. Listing 3.5 provides an example schema for the `@id` property, which will provide the user with auto-completion in the *GUI panel*.

```
"properties": {
  "@id": {
    "type": "string",
    "format": "uri",  # ensures that the input is a valid URI
    "metaConfigurator": {
      ontology: {
        mustBeUri: true,  # enables IRI auto-completion via SPARQL
        mustBeClassOrProperty: true  # if enabled: filters out results that are not classes/properties
      }
    }
  },
}
```

Listing 3.5: Schema for the @id property, using MetaConfigurator metadata for IRI-auto-completion.

3.5.3 Limitations

For IRI auto-completion to work, a SPARQL endpoint is required, with a graph that contains the relevant nodes. Also, depending on the query it can take multiple seconds until a response is retrieved and the auto-completion suggestions updated. Furthermore, in the schema editor not all JSON-LD keywords are supported by the GUI. For JSON-LD fields to appear in the data editor, they must be added to the schema by hand.

3.6 JSON Schema Diagram

This section is about representing a JSON schema in a graphical way, making it more intuitive for users to understand the structure of a schema. For *MetaConfigurator*, a novel technique is designed and implemented, which does not only create a graphical diagram out of a schema, but also it is interactive and allows for editing of the schema. In literature there exist approaches to visualize any type of JSON data or to convert a JSON schema into a UML class diagram (see section 2.3.3), however, both of these approaches have limitations when applied for JSON schema: An approach to visualize arbitrary JSON documents will not "know" about the specifics of JSON schema, and include redundant information in the graph. For instance, instead of listing all properties of a JSON schema object inside the object node, this generic approach will create an additional `properties` node and separate nodes for each of the properties, connecting them with edges, as shown in figure 2.5. In UML, on the other hand, there is no clear specification on how to represent some of the JSON schema keywords, such as `oneOf`. Furthermore, to enable making the schema visualization interactive, a conversion from JSON schema to a static picture does not suffice but instead a technique is required that can generate an internal graph representation (data structure) from a JSON schema, which can then be visualized and provided to the user in an interactive manner.

The schema in listing 3.6 serves as an example, based on which the diagrams and algorithm results in this section are presented. For the sake of readability and requiring less space on the page, YAML syntax is used. Semantically, it is identical to a JSON document.

3.6.1 Schema2Graph Algorithm

This section describes the technique *MetaConfigurator* uses to generate the schema graph.

Representation of the JSON Schema Features in the Diagram

Table 3.4 shows how the different JSON schema features are represented in the graph. Applying these mappings to the example schema from listing 3.6 results in a representation as shown in figure 3.22.

Generation of the Schema Graph

First, a recursive algorithm identifies all sub-schemas of the provided JSON schema documents and generates a so-called `Node` object for each. The algorithm is executed for the root schema, as well as for all entries in the `definitions` and `$defs` properties. Edges (see later steps) have `Nodes` as their source and target. In the next step, a for-loop over all sub-schema definitions is used to generate the rest of the graph data. It is required to perform this step after all sub-schemas have been identified and mapped to their corresponding paths, because handling attributes and edges may involve resolving references. For every sub-schema, which describes a JSON schema object, the object attributes are generated and added to the node. Furthermore, for all these sub-schemas the attribute edges are generated, as well as the edges for compositions and conditionals. For enumeration sub-schemas, the enumeration items are added to the node. All nodes and the generated edges are added to the graph. The main difficulty for these steps lies in resolving the proper attribute

Table 3.4: Corresponding diagram representation for JSON schema features.

JSON schema features / keywords	Diagram representation
`object`	Class.
`enum` or `const`	Enumeration.
`array`	No dedicated node for array, only the subschema will get one. Represented by multiplicity in the edges.
`properties`	All properties are listed as attribute of the class.
patternProperties	Same as for `properties`, using the pattern as attribute name.
additionalProperties	Composition edge to subschema (which is represented in separate node).
inline `object`, `enum` or `const` property	Composition edge to the class/enumeration.
referenced `object`, `enum` or `const` property	Aggregation edge to the class/enumeration.
`deprecated` property	Strikethrough in the attribute name.
`required` property	Red asterisk on the right of attribute name.
`allOf`	Specialization edge to the subschema (which is represented in separate node).
`oneOf`	Specialization edge to the subschema (which is represented in separate node), with the label `oneOf`.
`anyOf`	Specialization edge to the subschema (which is represented in separate node), with the label `anyOf`.
`if`	Association edge to the subschema (which is represented in separate node), with the label `if`.
`then`	Specialization edge to the subschema (which is represented in separate node), with the label `then`.
`else`	Specialization edge to the subschema (which is represented in separate node), with the label `else`.

type descriptions, as well as the proper receiver of an edge. If the sub-schema of an attribute is a `$ref`, the reference should be resolved. Also, if the sub-schema of an attribute specifies an array, the array item definition should be resolved and used instead of the array definition, while noting down that the attribute is of array type. This way, the graph will not show distinct nodes for each reference or array but instead connect the edges directly with the corresponding object definitions. Finally, the graph is ”trimmed”, by removing nodes that are neither connected to any edges nor do they correspond to a JSON schema object or enum.

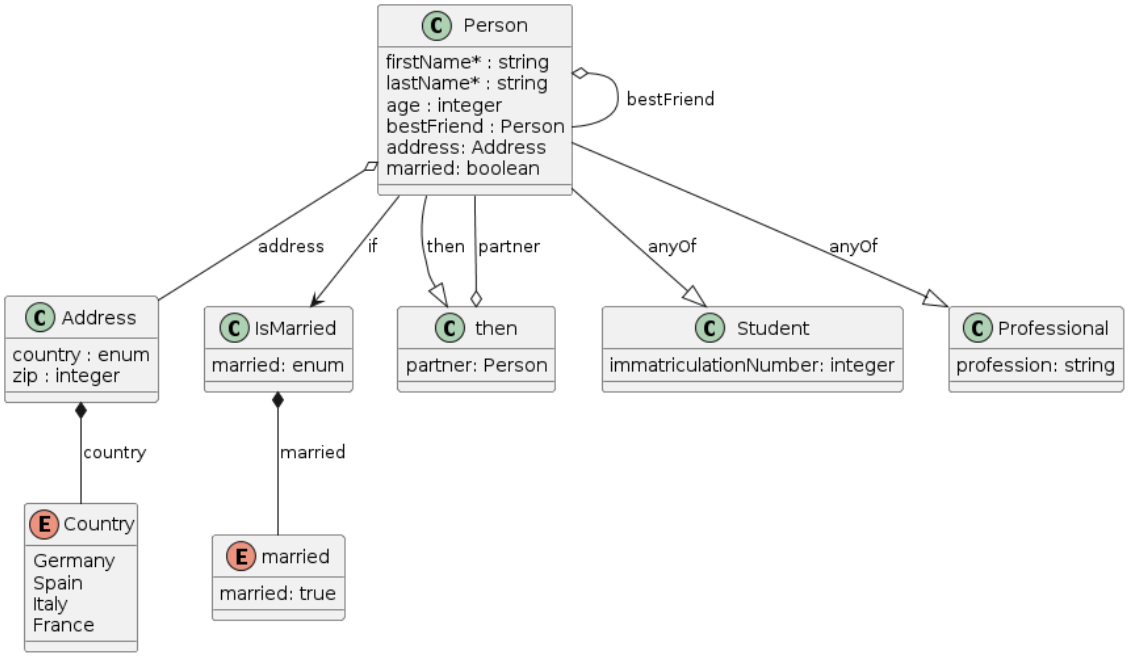

Figure 3.22: Schema diagram of the schema from listing 3.6 in an UML-like way.

3.6.2 Implementation and Results

For the diagram visualization in *MetaConfigurator*, the TypeScript library *Vue Flow* is used. The diagram graph is generated as described in section 3.6.1. Next, an algorithm is applied on all the nodes and edges of the graph to position them in an organized manner. To compute the layout, the library *dagre* (see section 2.3.5) is used.

Visualization

Figure 3.23 shows the resulting diagram created by *MetaConfigurator* based on the example schema. Because users might have multiple *MetaConfigurator* panels open simultaneously their schema diagram panel might be larger in height than in width. Therefore, an alternative vertical layout is introduced, as shown in figure 3.24. This becomes more relevant when objects have many properties and are not deeply nested. Note that the depicted diagrams and the example schema make use of several advanced JSON schema features, such as *composition*, *conditionals* and *cyclic dependencies*. More simple schemas result in a more simple schema diagram. Figure 3.25 shows an exemplary schema diagram without advanced schema features. Also, when using the vertical layout, the edge handles for the individual object properties are placed at the height of the properties themselves, showing which edge belongs to which attribute, as visible in figure 3.25. The schema diagram is capable of showing large and complex schemas too. Figure 3.26 shows an overview of a schema diagram of the *EnzymeML* schema. Note that *MetaConfigurator* has a customizable limit on how many properties of an object and how many items of an enum to display. When the limit is exceeded, the remaining properties or items are hidden and only shown when the corresponding node is selected. One of the biggest implementation challenges was resolving references for different

combinations of advanced JSON schema keywords (e.g., `oneOf` to the *reference* to an array of *references*, each referring to a *conditional*). Furthermore, multiplicity is not drawn at the front/back of edges. As all edges reach their target nodes in the center, having two or more incoming nodes would result in the multiplicity label overlapping and render it unreadable. Instead, the edge label contains a "`[]`" suffix if the property is an array property.

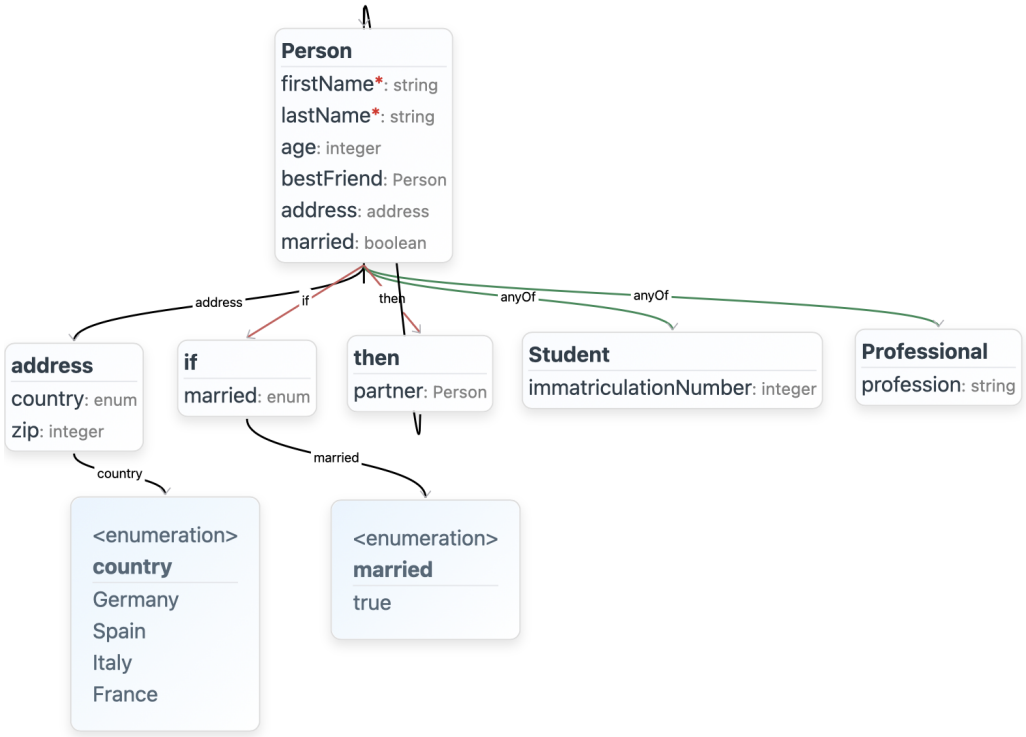

Figure 3.23: Schema diagram of the schema from listing 3.6 created by *MetaConfigurator*, with horizontal layout.

Navigation

The graph is interactive, and individual nodes can be moved. The view can also be moved and zoom applied. Individual nodes and also attributes can be selected, which is synchronized with the other panels of *MetaConfigurator* and selects them there too. Figure 3.27 shows the selection of an object attribute across the text editor, schema editor and GUI editor panels. Selected elements are highlighted. If the selected element is out of view, the view will automatically move to the element in a smooth transition. Because the user activity is synchronized across all panels, the user can switch between the different panels seamlessly. For example, they can navigate to a certain element by selecting it in the schema editor. The text editor will move to and focus on the same element, allowing the user to edit it.

Analogous to the *GUI editor* panel, the *schema editor* panel also allows traversal of the data tree. By double-clicking a node, *MetaConfigurator zooms* into it, showing only that node and its children and hiding the rest of the schema.

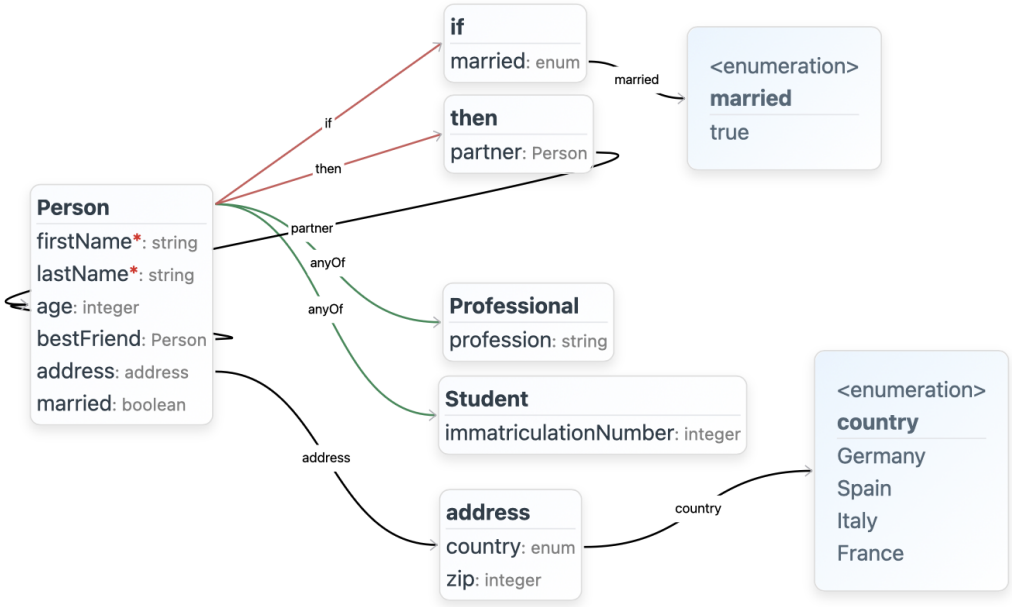

Figure 3.24: Schema diagram of the schema from listing 3.6 created by *MetaConfigurator*, with vertical layout.

Editing the Schema

The schema diagram can be used not only for visualizing the schema and for navigation, but also for making simple changes to the schema. While the *GUI panel* already makes editing schemas easier, for complete beginners who are not familiar with code or JSON-like data structures, it can still be complicated. To lower the entry barrier, schema editing is made possible in a graphical way using the schema diagram.

When a node is selected, it becomes editable. Figure 3.28 shows the `Occupation` object node from figure 3.25 being selected, with *1)* being a textfield to rename the object node, *2)* being a button to delete the node, and *3)* being a button to add new properties to the node.

When an object property is selected, the property itself becomes editable, as shown in figure 3.29. The property name can be changed (*1)*), the property type can be changed (*2)*), and the property can be removed (*3)*).

Figure 3.30 shows the `Occupation` node with a new `workAddress` property added. The user clicked at the type selection dropdown menu and gets to define the new type for the property. All simple types (`string`, `number`, `integer`, `boolean`, `null`), all non-inlined object schemas and arrays of all these types are suggested as options to the user. Non-inlined object schemas are all the object schemas inside the `$defs` or `definitions` section. For extracting inlined object schemas and moving them to the definitions section, a corresponding button is shown for all nodes where the object schema is inlined. After selecting the type, the edges in the graph are updated. If an object or enum is selected as type, a new edge is created from the property to the node of that object or enumeration. For `workAddress`, the type `Address` was selected, leading to the new graph as shown in figure 3.31. Enumerations can be edited similarly, except that the individual items do not have a type.

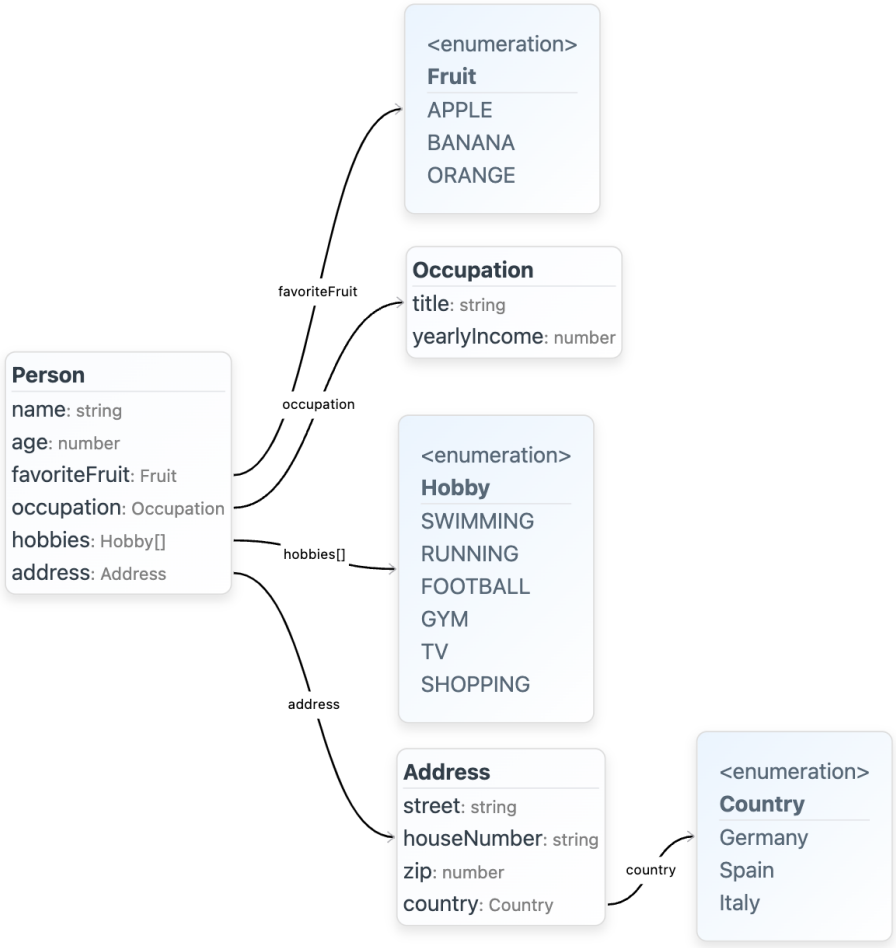

Figure 3.25: Schema diagram without advanced JSON schema features.

3.6.3 Limitations

The current schema diagram implementation has limitations. More advanced JSON schema keywords (*conditionals*, *compositions*, and other property *constraints*) can be visualized, but not modified with the *schema diagram* panel to avoid making it overwhelming. When working with advanced keywords, it is recommended to use the *schema diagram* panel and the *GUI* panel simultaneously. Second, only standard arrowheads are used instead of distinguishing between aggregation, association, specialization and so on.

In the future, a different graph visualization library could be used or the current visualization implementation in *MetaConfigurator* adjusted, to make the diagram look more UML-like, and use different arrowheads, as planned in the design in section 3.6.1. Finally, keyboard controls (e.g., copy and paste shortcut) could be added for increased accessibility.

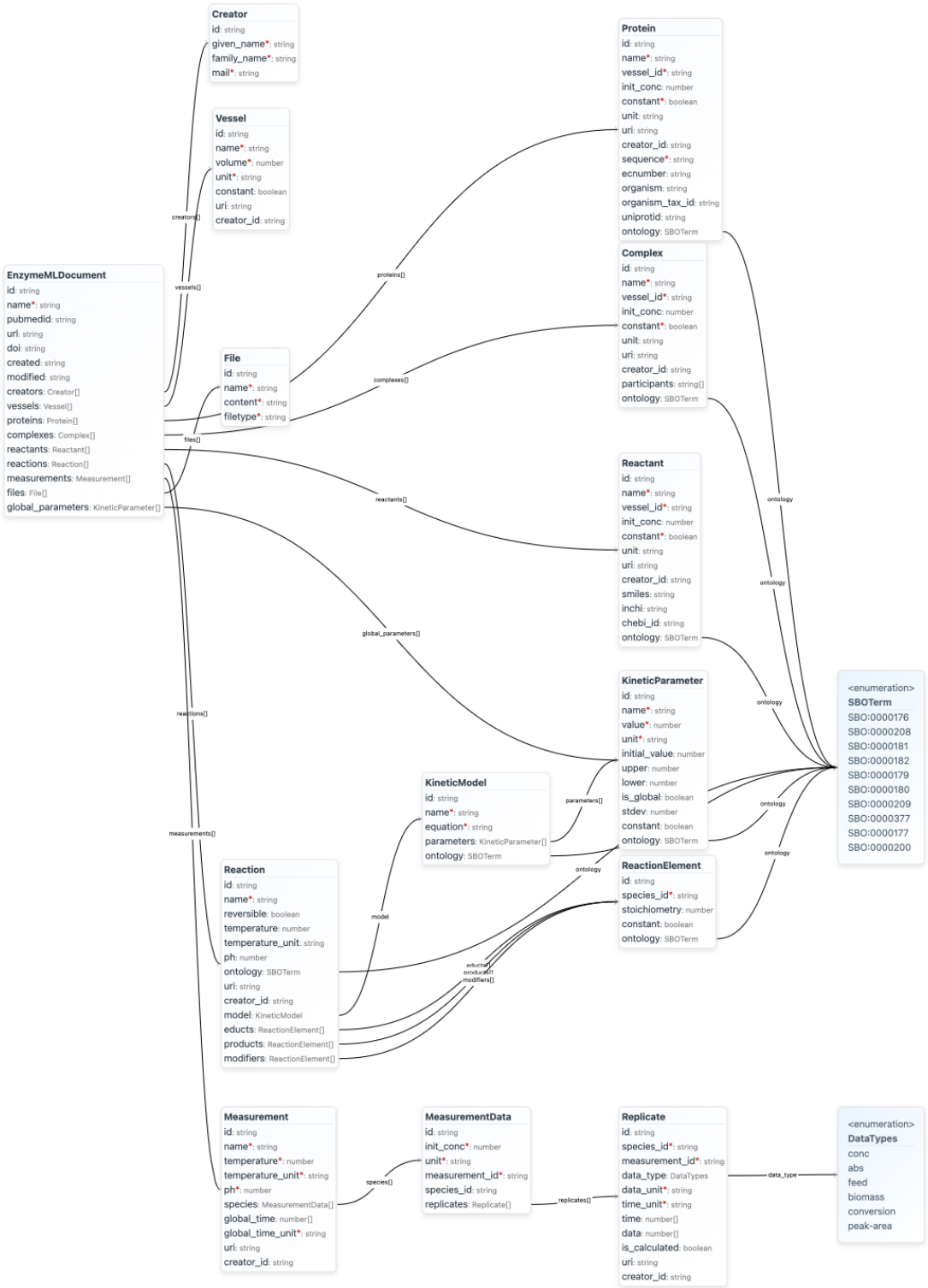

Figure 3.26: Schema diagram of the *EnzymeML* schema.

```yaml
title: Person Schema
type: object
required: ["firstName", "lastName"]
properties:
  firstName:
    type: string
  lastName:
    type: string
  age:
    type: integer
    minimum: 0
  bestFriend:
    $ref: "#"
  address:
    $ref: "#/$defs/address"
  married:
    type: boolean
$defs:
  address:
    title: Address
    type: object
    properties:
      country:
        type: string
        enum: ["Germany", "Spain", "Italy", "France"]
      zip:
        type: integer
if:
  type: object
  properties:
    married:
      type: boolean
      const: true
then:
  type: object
  properties:
    partner:
      $ref: "#"
anyOf:
  - type: object
    title: Student
    properties:
      immatriculationNumber:
        type: integer
  - type: object
    title: Professional
    properties:
      profession:
        type: string
```

Listing 3.6: JSON schema example using advanced keywords.

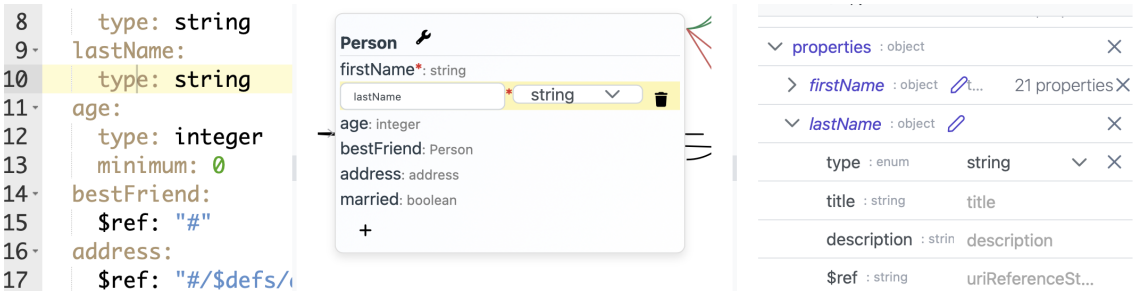

Figure 3.27: Synchronizing the user selection across all panels.

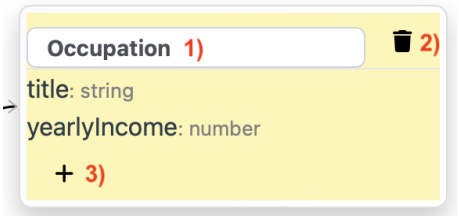

Figure 3.28: Schema diagram with `Occupation` object node selected.

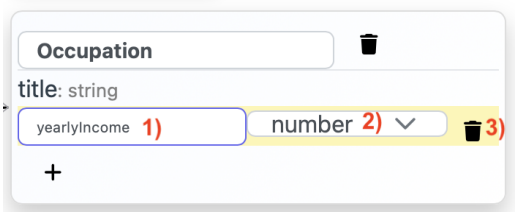

Figure 3.29: Schema diagram with object property `yearlyIncome` being selected.

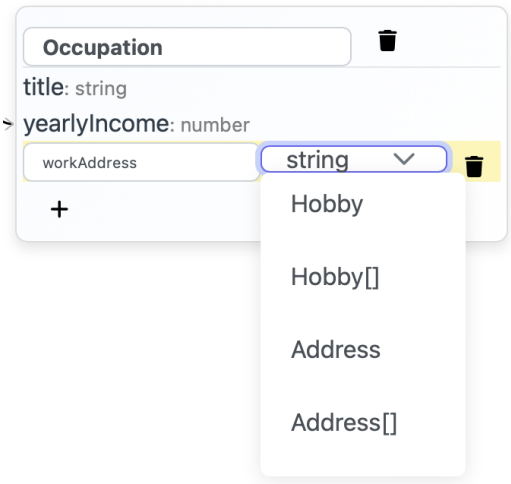

Figure 3.30: Schema diagram, where the user is selecting the type for property `workAddress`.

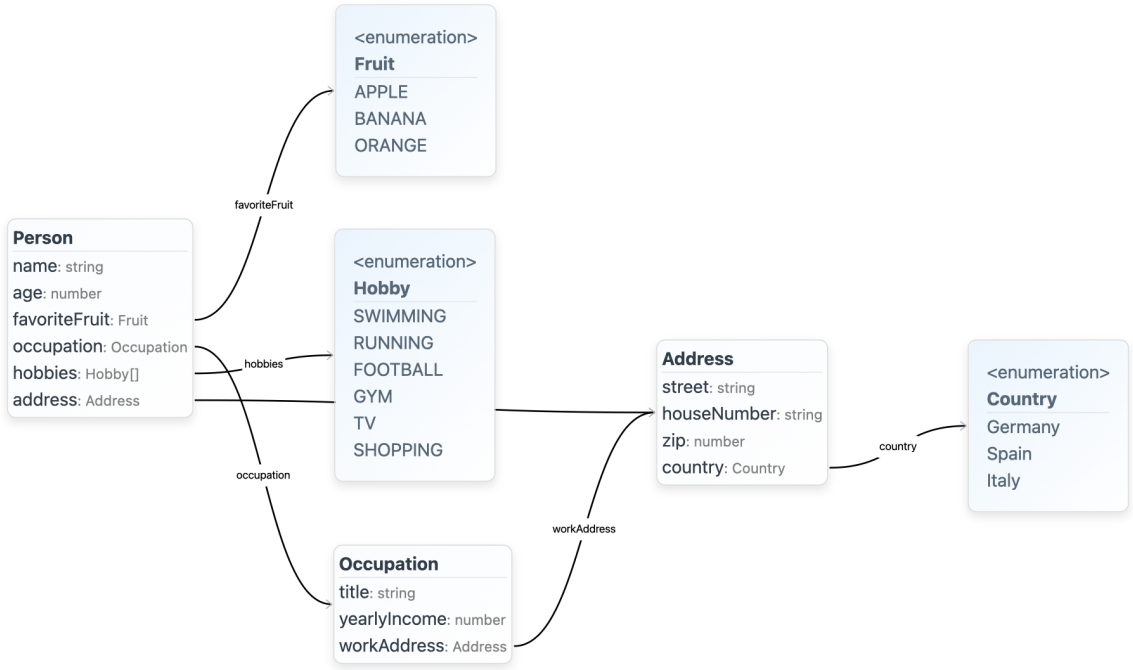

Figure 3.31: Resulting schema diagram after changing the type of property `workAddress`.

3.7 Miscellaneous Improvements

Furthermore, the following miscellaneous improvements have been performed:

- Implementation of full YAML support by applying the visitor pattern [87] on the YAML concrete syntax tree[101]. This enables synchronizing the element selected by the user in any panel with the selection in the YAML text editor.
- Making the *MetaConfigurator* user interface more consistent, for example, by replacing the radio buttons in the initial schema selection dialog with regular buttons and changing the labels of some user interface elements.
- Making it easier to understand the *Add item* button for arrays in the GUI: 1. by automatically expanding empty arrays when a schema is initially loaded, to make the *Add item* button instantly visible; and 2. by renaming the button label. If a `title` is defined for the `items` schema, then that title will be used in the label. For example, if the array is a list of creators and the schema `items` has the `title` `Creator`, then the button will get the label *Add Creator*. If no *title* is defined, then the button will use the array name in its label, for example *Add item (creators)*.
- Most schemas support additional properties that are unknown to the schema. To represent this in the GUI panel, an *Add property* is shown, allowing the user to add arbitrary additional data. While it is beneficial for schemas to allow additional properties, for compatibility and flexibility reasons, it is often not desired by the user to add any property unknown to the schema by hand. Instead, an *Add property* button in the GUI can even be confusing and misleading, creating the assumption that the created property will be of relevance for the schema. To avoid this confusion, without requiring any change in the schema, *MetaConfigurator* will now by default no longer show the *Add property* button for additional unknown properties, except the schema explicitly defines a schema for these additional properties, differing from `false` (no additional property allowed) and `true` (any arbitrary additional property allowed). This feature is optional and can be deactivated in the settings menu.
- When renaming the property of an object, the order of properties is now preserved. Previously, the renamed property was moved to the bottom.
- Newly added properties are no longer put last but, instead, put into the same order as defined by the schema.
- The tab size in the text editor was introduced as a setting that can be edited by the user.
- Shortened the notation of array items in the GUI panel. Previously, it was `arrayname[0]`, `arrayname[1]`, etc. Now it is `Item 1`, `Item 2`, etc.
- Introduced colors for the different modes (*data editor*: black, *schema editor*: green, *settings*) to visually distinguish between the modes. For example, the breadcrumb component in the *GUI panel* and *schema diagram* are colored based on their mode. This could be further extended, and the colors could be used for more components.

4 Application in Praxis

This chapter shows how the new functionality can be applied in praxis, demonstrating its applicability and impact.

4.1 preCICE Adapter Configuration Schema

preCICE is a coupling library for partitioned multi-physics simulations. It couples existing programs/solvers capable of simulating a subpart of the complete physics involved in a simulation [27]. Currently, it has more than 10 different configuration files for different adapters and tools. Some of these configurations are solver-specific, while others are more or less the same. They do not yet have a standard or a standard schema. Benjamin Uekermann, one of the main developers of *preCICE*, suggests introducing a common schema to enable more tooling support and also provides schema suggestions.[1] The standardization process to create and agree on a common schema will involve many different researchers and can take more than a year. Having an up-to-date web form (GUI) showing the fields of the schema, as well as the option to edit the schema, will be useful. This way, also the *preCICE* users who are not familiar with JSON schema can follow the process and understand the proposed file structure.

Using *MetaConfigurator*, a schema is created. See listing 4.1 for the initial schema proposal within the preCICE team. Furthermore, using the *snapshot and project sharing* functionality, a unique URL is generated, which will lead to *MetaConfigurator* with the *preCICE* schema loaded, and specific settings loaded. In the settings, the displayed title is changed from *MetaConfigurator* to `preCICE Adapter Configurator`, and the *schema* and *settings* tabs are disabled. The URL can be shared with users of *preCICE*, to edit their configuration files in an assisted manner. Figure 4.1 shows *MetaConfigurator* when accessing the URL. The schema and settings are pre-loaded.

[1] https://github.com/precice/preeco-orga/issues/18, accessed 2024/08/27.

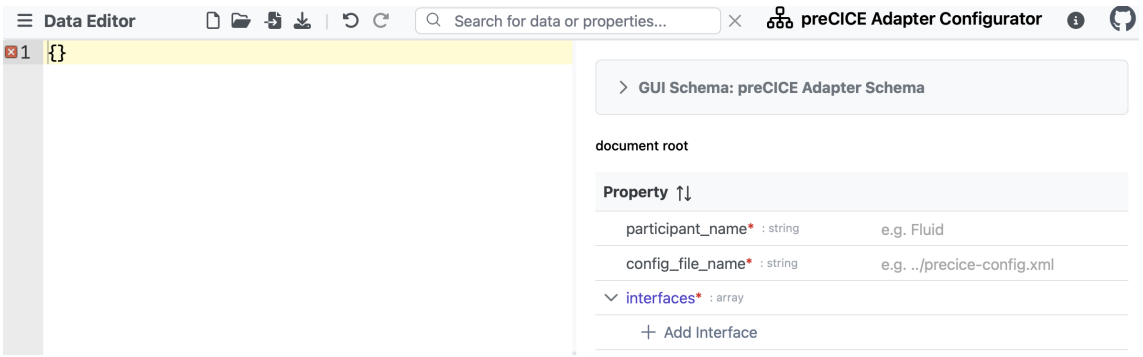

Figure 4.1: *MetaConfigurator* with custom title and preCICE adapter schema pre-loaded.

```yaml
$schema: http://json-schema.org/draft-07/schema#
type: object
properties:
  participant_name:
    title: Participant Name
    type: string
    description: Name of the participant.
    default: Fluid
  config_file_name:
    title: Configuration File Name
    type: string
    description: Path to the configuration file.
    default: ../precice-config.xml
  interfaces:
    type: array
    items:
      title: Interface
      type: object
      properties:
        mesh_name:
          title: Mesh Name
          type: string
          description: Name of the mesh associated with this interface.
          default: Fluid-Mesh
        write_data_names:
          type: array
          items:
            title: Write Data Field
            type: string
            default: Force
          description: List of data fields to be written on this mesh.
        read_data_names:
          type: array
          items:
            title: Read Data Field
            type: string
            default: Displacement
          description: List of data fields to be read on this mesh.
      required:
        - mesh_name
required:
  - participant_name
  - config_file_name
  - interfaces
title: preCICE Adapter Schema
```

Listing 4.1: Initial schema proposal for preCICE adapter configurations in YAML syntax.

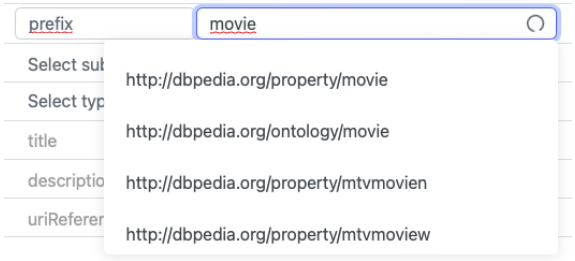

Figure 4.2: Ontology IRI auto-completion based in the `movie` search term.

4.2 Linking JSON Schema with DBpedia Ontology

This section provides an artificial example of how the ontology integration could be used. Our fictional user works in the movie critics industry. They manage a website where people can submit reviews about popular Hollywood movies. The user wants to apply machine learning on the review data to infer trends and other information. While their platform stores user reviews, it does not yet have detailed metadata about the movies themselves. Instead of entering all metadata by hand, they want to make use of the existing DBpedia knowledge graph, which stores information about movies, actors and more. The user wants to link their review data with the movie and actor nodes. This can be achieved using JSON-LD. First, in the *MetaConfigurator* settings, they enter the Dbpedia SPARQL endpoint. Then, the user creates a suitable *review data* schema in the *MetaConfigurator* schema editor. In the menubar, they enable the *show JSON-LD fields* option. Then they create a schema describing the review data. The main schema object is a `review`, which has a `movie` as a property. To link the review instances with Dbpedia movie entries, the `movie` property must be of type string. It can be further restricted by setting the `format` to `URI`. For the `movie` schema object, using the *MetaConfigurator* Ontology IRI auto-completion functionality, the user sees that the corresponding Dbpedia IRI is *http://dbpedia.org/ontology/movie* (figure 4.2).

To also get IRI auto-completion when entering review instances, each linked to a movie, the user adds `metaConfigurator/ontology` metadata to the movie property schema. Listing 4.2 shows the created schema in the YAML format. Figure 4.3 shows the resulting *data editor* view, where the user can enter review instances and will get movie IRI auto-completion.

```yaml
type: object
properties:
  reviews:
    type: array
    items:
      type: object
      properties:
        reviewer:
          type: string
        rating:
          type: number
          maximum: 10
          minimum: 0
        message:
          type: string
          maxLength: 300
        movie:
          type: string
          format: uri
          "@id": http://dbpedia.org/ontology/movie
          metaConfigurator:
            ontology:
              mustBeUri: true. # To enable the auto-completion
              mustBeClassOrProperty: false. # Otherwise only Ontology classes and properties
are shown but not movie instances
```

Listing 4.2: Movie review schema, linking movies to Dbpedia IRIs.

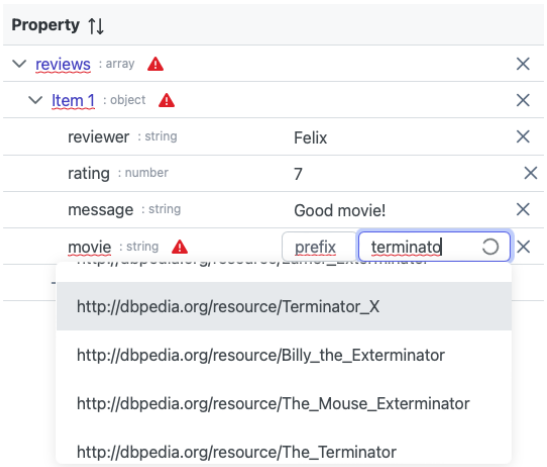

Figure 4.3: Data editor GUI for movie review schema with IRI auto-completion for movies.

4.3 Application in Biochemistry

The *Institute of Biochemistry and Technical Biochemistry* of University of Stuttgart stores some of their research data in Excel files, which differ in their structure from researcher to researcher. They want to introduce common standards for their data to be able to apply automated scripts, machine learning, and in the future also integrate the data into a knowledge graph. Hence, they are interested in moving from Excel to a machine-readable data format, such as JSON and to use tooling for creating a data model (schema). Furthermore, as most Biochemistry researchers are not software developers or experienced with JSON, and especially not JSON schema, the tooling for creating schemas should be simple to use and hide the complexities of JSON schema from the user. *MetaConfigurator* is introduced to the researchers, and in a joint effort, their needs and wishes are collected, and *MetaConfigurator* is extended accordingly. Mainly, the *CSV Import* and *Graphical Schema Editing* were requested by them.

This section describes an exemplary use case, developed jointly, where *MetaConfigurator* is used to migrate Excel data into a JSON document and infer the corresponding schema. The use case is to automatically extend the given data from a list of synthesis experiments. For every compound in the list, the PubChem database is queried and the `inchi_code`, `smiles_code` and `molecular_weight` are retrieved. This additional metadata is appended to the JSON document. Using the graphical schema editor, the schema is adjusted to include the new fields. An external Python script is developed which automatically appends the additional metadata to the JSON document. It is demonstrated that every step within *MetaConfigurator* can be performed without prior JSON or JSON schema knowledge. The complete step-by-step instructions, input files and Python script can be found on GitHub.[2] The Python script was developed jointly with Torsten Gieß from the Biochemistry department.

4.3.1 Importing the data

First, the Excel table is exported as a CSV document named `ec-mof-synthesis.csv`. In the repository, the CSV file is already provided. Next, in the *MetaConfigurator* data editor tab, the menu bar option *Import Data...* →*Import CSV Data* is selected. In the import dialog, the `ec-mof-synthesis.csv` file is uploaded. *MetaConfigurator* automatically detects that the file uses a *comma* as delimiter and a *dot* as decimal separator. The option *Independent Table* is selected by default and remains unchanged. Also, the user keeps the option *Infer and generate schema for the data* selected (figure 4.4). The user could modify the mappings from CSV to JSON for the individual CSV columns, but can also leave them unchanged because the default suggestions are suitable. Finally, the user presses the *Import* button (figure 4.5), resulting in the CSV being read and converted to a JSON document and the corresponding schema being auto-generated. Figure 4.7 shows the imported data, and figure 4.6 shows the resulting schema in the diagram view.

[2] https://github.com/MetaConfigurator/meta-configurator/tree/main/examples/biochemistry, accessed 2024/08/27.

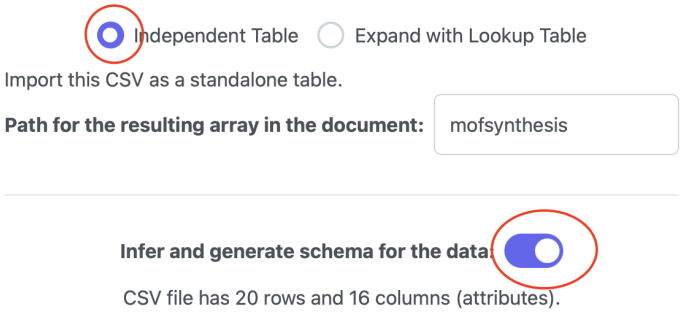

Figure 4.4: Import Options for `ec-mof-synthesis.csv`.

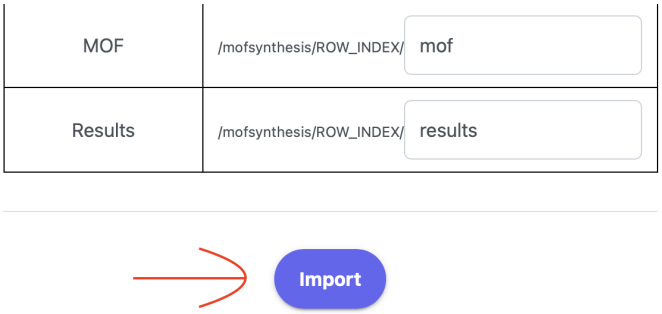

Figure 4.5: Attribute mappings and `import` button for `ec-mof-synthesis.csv`.

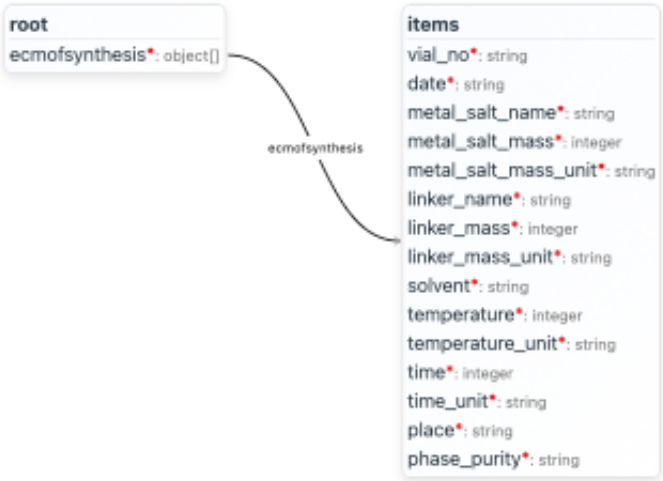

Figure 4.6: Generated Schema for imported `ec-mof-synthesis.csv`.

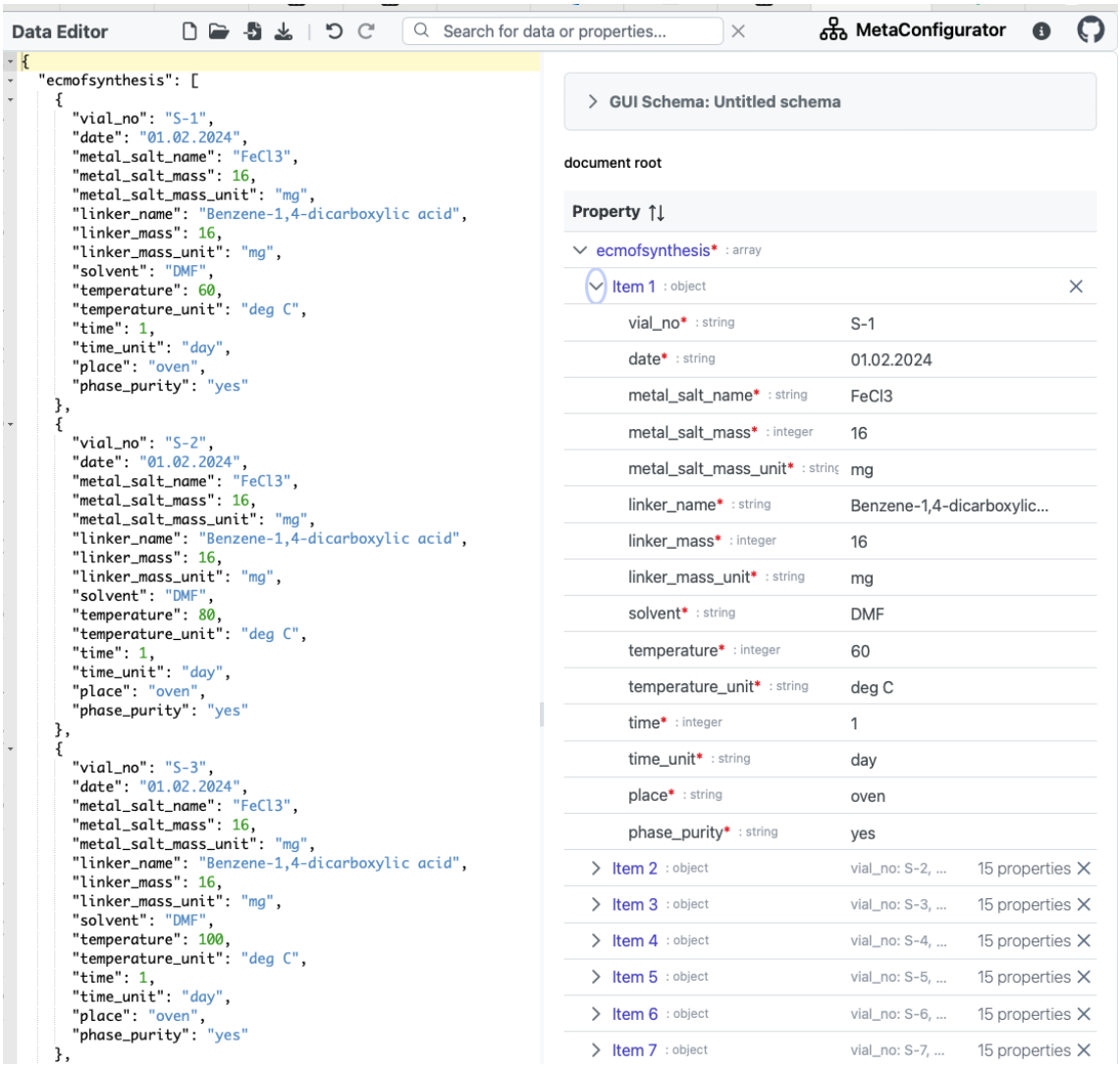

Figure 4.7: `ec-mof-synthesis.csv` imported and converted to JSON.

4.3.2 Improving the schema

There are multiple ways the schema can be improved. For example, the user can introduce the enums `weight_unit`, `mass_unit` and `time_unit` and make use of them, instead of allowing any string for the properties which represent a mass, temperature or time unit. All these changes can be performed with a few clicks by using the diagram view. Figure 4.8 shows the adjusted schema.

4.3.3 Python script for appending metadata and adjusted schema

The user prepares a Python script to enrich the synthesis data JSON document with additional metadata. Both the `metal_salt` and `linker` are chemical compounds. The database PubChem allows querying information about compounds. The user uses it to query the `inchi_code`, `smiles_code` and `molecular_weight` for every `metal_salt` and `linker`. The compound structure can be represented by

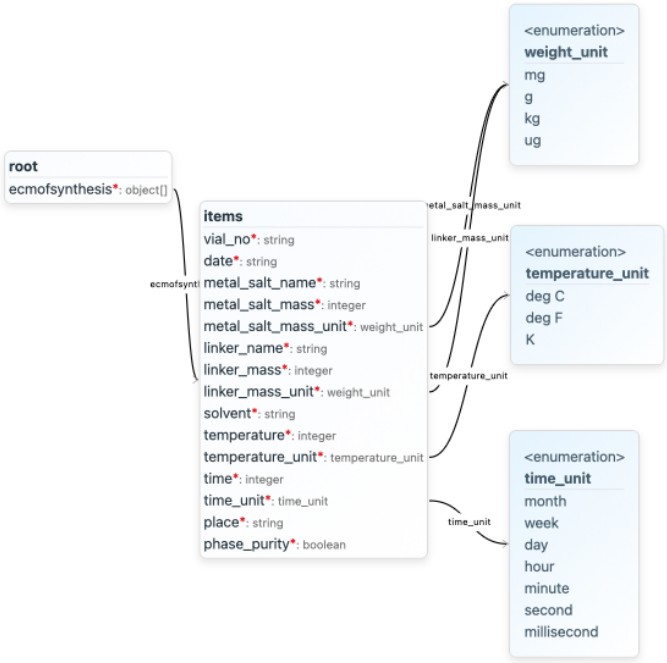

Figure 4.8: Schema after adjustments were performed by the user using the interactive diagram view.

an object node, which is added by the user. Two new properties of the new `compound` type are added to the `items` object: `metal_salt` and `linker`. Figure 4.9 shows the resulting schema. Note that the design of the data model itself is up to the preference and use case of the user. In the example, we add new properties, without removing or changing existing ones. An alternative could be to also move the compound `mass` and `mass_unit` into the `compound` node. Figure 4.10 shows the resulting data after running the Python script to enrich the JSON document. The *GUI panel* properly shows the `linker` and `metal_salt` objects, because the schema had been adjusted accordingly.

4.3.4 Result

Using *MetaConfigurator*, it is possible to port Excel tables into JSON, making them more machine-readable. The schema can be inferred automatically and adjusted in a graphical UI, without prior JSON or JSON schema experience being required. Having a schema allows communicating the data model, sharing it with others, or even to technically enforce it (schema validation). Furthermore, having the data in JSON format makes it easy to apply tooling, such as the Python script, to enrich the data with more information.

Figure 4.9: Schema with the `compound` type added.

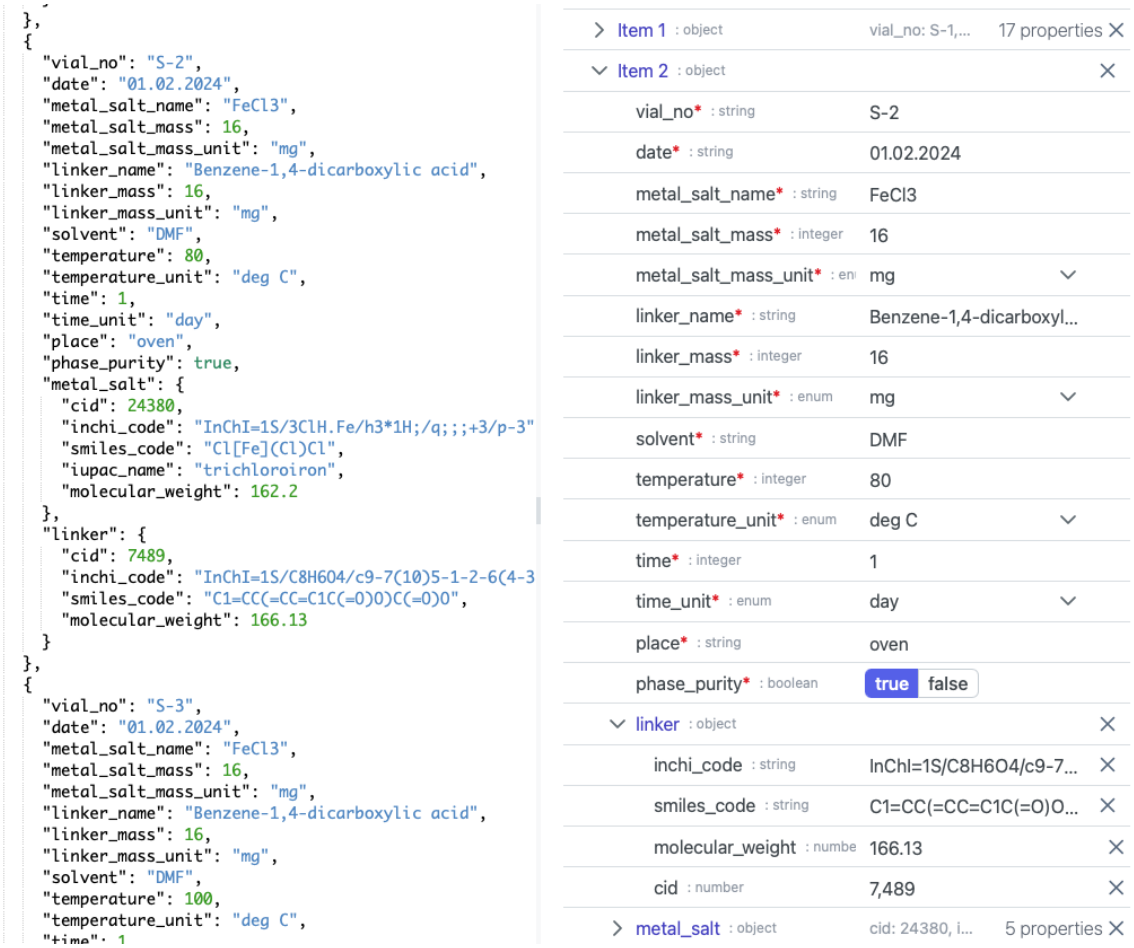

Figure 4.10: Enriched synthesis data in the *MetaConfigurator data editor* tab.

5 Conclusion and Outlook

Data models are essential for ensuring structured, consistent, and interoperable data across various systems. Creating standard data models allows for effective collaboration and integration across domains and for data validation. For instance, the biochemistry researchers from University of Stuttgart work to standardize and structure their experimental data, while the developers of the *preCICE* coupling library aim to create a shared data model for managing their complex simulation environments.

Tools can significantly aid in the creation and maintenance of these data models. *MetaConfigurator*, designed as a schema editor and form generator for JSON and YAML documents, is one such tool. Based on real-world use cases and discussions with users, several improvements were designed and implemented:

1. **A more user-friendly schema editor**, offering an easy mode for beginners and an advanced mode for experts, lowering the entry barrier for users with different levels of experience.
2. **A CSV import feature**, allowing a seamless transition from Excel data to the JSON format, with automatic schema inference to save time and avoid manual work.
3. **Snapshot sharing functionality**, enabling users to quickly share data models or schemas with others, fostering collaboration and easy communication.
4. **Ontology integration**, which provides auto-completion for URIs using SPARQL queries, helping users link their data models to existing ontologies and facilitating the creation of knowledge graphs.
5. **A graphical schema editor**, which visualizes schemas in a diagram-like interface, similar to UML class diagrams, allowing users to edit schemas visually without needing to dive into the technical specifics of JSON schema.

These enhancements have already been applied successfully. In biochemistry, researchers at the University of Stuttgart transitioned from using Excel to managing their data in JSON format with minimal effort, thanks to *MetaConfigurator's* automatic schema inference and graphical editing features. The tool also demonstrates potential in the *preCICE* project, where it will be used in an upcoming standardization process, helping developers coordinate and create a unified data model for their simulation library.

5.1 Outlook

While *MetaConfigurator* has evolved into a more powerful and user-friendly tool, there is still room for improvement. Future work could further refine the ontology integration, possibly by developing an algorithm that generates JSON schemas directly from existing ontologies. A knowledge graph

visualization could be integrated, for users who make use of JSON-LD. The graphical schema editor could be enhanced with more advanced features such as keyboard support (e.g., copy-paste functionality via shortcuts). It could also be improved by making the diagram more UML-like and distinguishing between different arrowheads. Also, more JSON schema specific utility functions could be provided to the user, such as a schema *satisfiability* check [1]. For the snapshot sharing functionality, legal constraints and the University guidelines must be clarified before the backend can be made available for the public. Another big extension could be the integration of artificial intelligence. A text field could be added to the UI, where the user can enter prompts to transform the schema or data according to their prompts written in natural language. The user prompts could be preprocessed and then, together with the current schema or data, fed into an Large Language Model (LLM) [99], which will return an updated schema or data document. The response of the LLM could be automatically postprocessed to ensure it resembles a valid JSON or YAML document and adheres to the schema. Another promising field could be the generation of source code (e.g. Python classes) based on a schema. Furthermore, to attract and satisfy a larger audience and more active users and collaborators, more documentation should be created and maintained. Also, a bigger user study could be conducted to gain more insights about how users with different levels of JSON experience engage and interact with the tool and which aspects should be further improved. By continuing to build on these advancements, *MetaConfigurator* can maintain its role as an indispensable tool for simplifying data model creation and fostering collaboration across diverse fields.

Bibliography

[1] L. Attouche, M.-A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, S. Scherzinger. *Witness Generation for JSON Schema*. 2022. arXiv: 2202.12849 [cs.DB]. URL: https://arxiv.org/abs/2202.12849 (cit. on p. 74).

[2] M.-A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani. "Parametric schema inference for massive JSON datasets". In: *The VLDB Journal* 28 (2019), pp. 497–521 (cit. on p. 37).

[3] M.-A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani. "Schemas and types for JSON data: from theory to practice". In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 2060–2063 (cit. on p. 37).

[4] M.-A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, S. Scherzinger. *An Empirical Study on the "Usage of Not" in Real-World JSON Schema Documents (Long Version)*. 2021. arXiv: 2107.08677 [cs.DB] (cit. on pp. 13, 15, 32).

[5] M.-A. Baazizi, H. B. Lahmar, D. Colazzo, G. Ghelli, C. Sartiani. "Schema inference for massive JSON datasets". In: *Extending Database Technology (EDBT)*. 2017 (cit. on p. 37).

[6] J. W. Backus. "The syntax and the semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference". In: *ICIP Proceedings*. 1959, pp. 125–132 (cit. on p. 9).

[7] A. Bajaj, J. Knight. "User Interface Generation from the Data Schema". In: *Systems Analysis and Design for Advanced Modeling Methods: Best Practices*. IGI Global, 2009, pp. 145–153 (cit. on p. 14).

[8] H. Bär, R. Hochstrasser, B. Papenfuß. "SiLA: Basic standards for rapid integration in laboratory automation". In: *Journal of laboratory automation* 17.2 (2012), pp. 86–95 (cit. on p. 28).

[9] W. Barth, M. Jünger, P. Mutzel. "Simple and efficient bilayer cross counting". In: *Graph Drawing: 10th International Symposium, GD 2002 Irvine, CA, USA, August 26–28, 2002 Revised Papers 10*. Springer. 2002, pp. 130–141 (cit. on p. 21).

[10] D. Beckett, jan grant. *RDF Test Cases*. W3C Recommendation. https://www.w3.org/TR/2004/REC-rdf-testcases-20040210/. W3C, Feb. 2004 (cit. on p. 23).

[11] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, G. Carothers. "RDF 1.1 Turtle". In: *World Wide Web Consortium* (2014), pp. 18–31 (cit. on p. 23).

[12] O. Ben-Kiki, C. Evans, I. döt Net. "YAML ain't markup language version 1.2". In: *Available on: http://yaml. org/spec/1.2/spec. html* (2009) (cit. on p. 9).

[13] T. Berners-Lee, R. Fielding, L. Masinter. *Uniform resource identifier (URI): Generic syntax*. Tech. rep. 2005 (cit. on p. 23).

[14] J. Bosak, T. Bray, D. Connolly, E. Maler, G. Nicol, M. Sperberg-McQueen, L. Wood, J. Clark. *W3C XML Specification DTD ("XMLspec")*. 1998. URL: https://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm (cit. on p. 13).

[15] U. Brandes, B. Köpf. "Fast and simple horizontal coordinate assignment". In: *International Symposium on Graph Drawing*. Springer. 2001, pp. 31–44 (cit. on p. 21).

[16] R. T. Bree, G. Gallagher. "Using Microsoft Excel to code and thematically analyse qualitative data: a simple, cost-effective approach." In: *All Ireland Journal of Higher Education* 8.2 (2016) (cit. on p. 35).

[17] R. G. Brereton. *Chemometrics: data analysis for the laboratory and chemical plant*. John Wiley & Sons, 2003 (cit. on p. 35).

[18] D. Brickley, R. Guha. *RDF Schema 1.1*. W3C Recommendation. https://www.w3.org/TR/2014/REC-rdf-schema-20140225/. W3C, Feb. 2014 (cit. on p. 25).

[19] D. Brickley, L. Miller. *FOAF vocabulary specification 0.91*. 2007 (cit. on p. 23).

[20] K. W. Broman, K. H. Woo. "Data organization in spreadsheets". In: *The American Statistician* 72.1 (2018), pp. 2–10 (cit. on p. 35).

[21] U. Carion. *JSON Type Definition*. RFC 8927. Nov. 2020. DOI: 10.17487/RFC8927. URL: https://www.rfc-editor.org/info/rfc8927 (cit. on p. 13).

[22] G. Carothers. *RDF 1.1 N-Quads*. W3C Recommendation. https://www.w3.org/TR/2014/REC-n-quads-20140225/. W3C, Feb. 2014 (cit. on p. 23).

[23] N. Catenazzi, L. Sommaruga, R. Mazza. "User-Friendly Ontology Editing and Visualization Tools: The OWLeasyViz Approach". In: *2009 13th International Conference Information Visualisation*. 2009, pp. 283–288. DOI: 10.1109/IV.2009.34 (cit. on p. 27).

[24] P. P.-S. Chen. "The entity-relationship model—toward a unified view of data". In: *ACM transactions on database systems (TODS)* 1.1 (1976), pp. 9–36 (cit. on p. 18).

[25] P. P.-S. Chen. "The entity-relationship model: a basis for the enterprise view of data". In: *Proceedings of the June 13-16, 1977, National Computer Conference*. AFIPS '77. Dallas, Texas: Association for Computing Machinery, 1977, pp. 77–84. ISBN: 9781450379144. DOI: 10.1145/1499402.1499421. URL: https://doi.org/10.1145/1499402.1499421 (cit. on p. 18).

[26] N. Chomsky. "Three models for the description of language". In: *IRE Transactions on information theory* 2.3 (1956), pp. 113–124 (cit. on p. 9).

[27] G. Chourdakis, K. Davis, B. Rodenberg, M. Schulte, F. Simonis, B. Uekermann, G. Abrams, H. Bungartz, L. Cheung Yau, I. Desai, K. Eder, R. Hertrich, F. Lindner, A. Rusch, D. Sashko, D. Schneider, A. Totounferoush, D. Volland, P. Vollmer, O. Koseomur. "preCICE v2: A sustainable and user-friendly coupling library [version 2; peer review: 2 approved]". In: *Open Research Europe* 2.51 (2022). DOI: 10.12688/openreseurope.14445.2. URL: https://doi.org/10.12688/openreseurope.14445.2 (cit. on pp. 11, 61).

[28] P. Čontoš, M. Svoboda. "JSON schema inference approaches". In: *International Conference on Conceptual Modeling*. Springer. 2020, pp. 173–183 (cit. on p. 37).

[29] D. F. Cook, C. W. Zobel, Q. J. Nottingham. "Excel-based application of data visualization techniques for process monitoring in the forest products industry." In: *Forest products journal* 54.5 (2004) (cit. on p. 35).

[30] D. Crockford. “The application/json Media Type for JavaScript Object Notation (JSON)”. In: *Internet Engineering Task Force IETF Request for Comments* (2006) (cit. on p. 9).

[31] S. Decker, S. Melnik, F. Van Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann, I. Horrocks. “The semantic web: The roles of XML and RDF”. In: *IEEE Internet computing* 4.5 (2000), pp. 63–73 (cit. on p. 22).

[32] L. Deligiannidis, K. J. Kochut, A. P. Sheth. “RDF data exploration and visualization”. In: *Proceedings of the ACM first workshop on CyberInfrastructure: information management in eScience*. 2007, pp. 39–46 (cit. on p. 27).

[33] J. Díaz, J. Petit, M. Serna. “A survey of graph layout problems”. In: *ACM Computing Surveys (CSUR)* 34.3 (2002), pp. 313–356 (cit. on p. 21).

[34] C. Draxl, M. Scheffler. “NOMAD: The FAIR concept for big data-driven materials science”. In: *Mrs Bulletin* 43.9 (2018), pp. 676–682 (cit. on p. 9).

[35] M. J. Dürst, M. Suignard. *Internationalized Resource Identifiers (IRIs)*. RFC 3987. Jan. 2005. DOI: 10.17487/RFC3987. URL: https://www.rfc-editor.org/info/rfc3987 (cit. on p. 23).

[36] H.-E. Eriksson, M. Penker, B. Lyons, D. Fado. *UML 2 toolkit*. John Wiley & Sons, 2003 (cit. on p. 19).

[37] D. Fallside, P. Walmsley. *XML Schema Part 0: Primer Second Edition - W3C Recommendation 28 October 2004*. 2004. URL: https://www.w3.org/TR/2004/REC-xmlschema-0-20041028/ (cit. on p. 13).

[38] J. Fenech. “A semantic editor: generation of a web-based user interface from an XML schema definition”. B.S. thesis. University of Malta, 2008 (cit. on p. 14).

[39] F. Frasincar, A. Telea, G.-J. Houben. “Adapting graph visualization techniques for the visualization of RDF data”. In: *Visualizing the Semantic Web: XML-based Internet and information visualization* (2006), pp. 154–171 (cit. on p. 27).

[40] M. S. Gal, D. L. Rubinfeld. “Data standardization”. In: *NYUL Rev.* 94 (2019), p. 737 (cit. on p. 9).

[41] E. R. Gansner, E. Koutsofios, S. C. North, K.-P. Vo. “A technique for drawing directed graphs”. In: *IEEE Transactions on Software Engineering* 19.3 (1993), pp. 214–230 (cit. on p. 21).

[42] G. Gauglitz. “Lab 4.0: sila or opc ua”. In: *Analytical and Bioanalytical Chemistry* 410.21 (2018), pp. 5093–5094 (cit. on p. 28).

[43] J. Golbeck, G. Fragoso, F. Hartel, J. Hendler, J. Oberthaler, B. Parsia. “The national cancer institute’s thesaurus and ontology”. In: *Journal of web semantics* 1.1 (2003), pp. 75–80 (cit. on p. 28).

[44] V. Grlickỳ. “An overview of RDF model representation formats”. In: *IIT. SRC 2005: Student Research Conference*. Citeseer. 2005, p. 242 (cit. on p. 23).

[45] J. Gruber. “Markdown: Syntax”. In: *URL http://daringfireball. net/projects/markdown/syntax. Retrieved on June* 24 (2012), p. 640 (cit. on p. 28).

[46] N. Guarino, D. Oberle, S. Staab. “What is an ontology?” In: *Handbook on ontologies* (2009), pp. 1–17 (cit. on pp. 21, 22).

[47] H. Guerrero, R. Guerrero, Rauscher. *Excel data analysis*. Springer, 2019 (cit. on p. 35).

[48] S. Harris, A. Seaborne. *SPARQL 1.1 Query Language*. W3C Recommendation. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/. W3C, Mar. 2013 (cit. on p. 25).

[49] N. Hartl, E. Wössner, Y. Sure-Vetter. "Nationale forschungsdateninfrastruktur (NFDI)". In: *Informatik Spektrum* 44.5 (2021), pp. 370–373 (cit. on p. 9).

[50] T. Heath. "Linked data: Evolving the Web into a global data space". In: *Morgan & Claypool* (2011) (cit. on p. 10).

[51] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. D. Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier, et al. "Knowledge graphs". In: *ACM Computing Surveys (Csur)* 54.4 (2021), pp. 1–37 (cit. on p. 22).

[52] *JSON Schema — json-schema.org*. https://json-schema.org. [Accessed 27-09-2024] (cit. on pp. 10, 13).

[53] M. Jünger, P. Mutzel. "2-Layer Straightline Crossing Minimization: Performance of Exact and Heuristic Algorithms". In: *Journal of Graph Algorithms and Applications* 1 (Jan. 1997). DOI: 10.1142/9789812777638_0001 (cit. on p. 21).

[54] E. Kajáti, M. Miškuf, P. Papcun. "Advanced analysis of manufacturing data in Excel and its Add-ins". In: *2017 IEEE 15th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*. IEEE. 2017, pp. 000491–000496 (cit. on p. 35).

[55] C. Kappestein. *Typeschema*. 2023. URL: https://typeschema.org/ (cit. on p. 13).

[56] J. Kasarda, M. Nečaskỳ, T. Bartoš. "Generating xforms from an xml schema". In: *Networked Digital Technologies: Second International Conference, NDT 2010, Prague, Czech Republic, July 7-9, 2010. Proceedings, Part II 2*. Springer. 2010, pp. 706–714 (cit. on p. 14).

[57] H. Knublauch, D. Kontokostas. *Shapes Constraint Language (SHACL)*. W3C Recommendation. https://www.w3.org/TR/2017/REC-shacl-20170720/. W3C, July 2017 (cit. on p. 26).

[58] D. E. Knuth. "Semantics of context-free languages". In: *Mathematical systems theory* 2.2 (1968), pp. 127–145 (cit. on p. 9).

[59] V. Koutkias. "From data silos to standardized, linked, and FAIR data for pharmacovigilance: current advances and challenges with observational healthcare data". In: *Drug Safety* 42.5 (2019), pp. 583–586 (cit. on p. 10).

[60] Y.-S. Kuo, N. Shih, L. Tseng, H.-C. Hu. "Generating form-based user interfaces for XML vocabularies". In: *Proceedings of the 2005 ACM symposium on Document engineering*. 2005, pp. 58–60 (cit. on p. 14).

[61] M. Lanthaler, D. Wood, R. Cyganiak. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/. W3C, Feb. 2014 (cit. on p. 23).

[62] S. Lauterbach, H. Dienhart, J. Range, S. Malzacher, J.-D. Spöring, D. Rother, M. F. Pinto, P. Martins, C. E. Lagerman, A. S. Bommarius, et al. "EnzymeML: Seamless data flow and modeling of enzymatic data". In: *Nature methods* 20.3 (2023), pp. 400–402 (cit. on p. 28).

[63] *Leading Serialization Format for Record Data*. [Accessed 14-06-2023]. URL: https://www.ibm.com/topics/avro (cit. on p. 13).

[64] D. Lee, W. W. Chu. “Comparative Analysis of Six XML Schema Languages”. In: *SIGMOD Rec.* 29.3 (Sept. 2000), pp. 76–87. ISSN: 0163-5808. DOI: 10.1145/362084.362140. URL: https://doi.org/10.1145/362084.362140 (cit. on p. 13).

[65] S.-H. Leitner, W. Mahnke. “OPC UA–service-oriented architecture for industrial applications”. In: (2006) (cit. on p. 28).

[66] S. Lohmann, V. Link, E. Marbach, S. Negru. “WebVOWL: Web-based visualization of ontologies”. In: *Knowledge Engineering and Knowledge Management: EKAW 2014 Satellite Events, VISUAL, EKM1, and ARCOE-Logic, Linköping, Sweden, November 24-28, 2014. Revised Selected Papers. 19*. Springer. 2015, pp. 154–158 (cit. on p. 27).

[67] M. Maeder, Y.-M. Neuhold. *Practical data analysis in chemistry*. Elsevier, 2007 (cit. on p. 35).

[68] T. Marrs. *JSON at work: practical data integration for the web*. O’Reilly Media, Inc., 2017, pp. 269–286 (cit. on p. 13).

[69] W. Martens, F. Neven, M. Niewerth, T. Schwentick. “BonXai: Combining the Simplicity of DTD with the Expressiveness of XML Schema”. In: *ACM Trans. Database Syst.* 42.3 (Aug. 2017). ISSN: 0362-5915. DOI: 10.1145/3105960. URL: https://doi.org/10.1145/3105960 (cit. on p. 13).

[70] B. Motik, B. Parsia, P. Patel-Schneider. *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition)*. W3C Recommendation. https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/. W3C, Dec. 2012 (cit. on p. 27).

[71] R. A. Muenchen. “The popularity of data analysis software”. In: *URL http://r4stats. com/popularity* (2012) (cit. on p. 35).

[72] F. Neubauer, P. Bredl, M. Xu, K. Patel, J. Pleiss, B. Uekermann. “MetaConfigurator: A User-Friendly Tool for Editing Structured Data Files”. In: *Datenbank-Spektrum* (2024), pp. 1–9. ISSN: 1618-2162. DOI: 10.1007/s13222-024-00472-7 (cit. on pp. 10, 13, 16, 32).

[73] J. L. Neyeloff, S. C. Fuchs, L. B. Moreira. “Meta-analyses and Forest plots using a microsoft excel spreadsheet: step-by-step guide focusing on descriptive data analysis”. In: *BMC research notes* 5 (2012), pp. 1–6 (cit. on p. 35).

[74] C. North, N. Conklin, V. Saini. “Visualization schemas for flexible information visualization”. In: *IEEE Symposium on Information Visualization, 2002. INFOVIS 2002.* IEEE. 2002, pp. 15–22 (cit. on p. 27).

[75] OECD. *Data-Driven Innovation*. 2015, p. 456. DOI: https://doi.org/https://doi.org/10.1787/9789264229358-en. URL: https://www.oecd-ilibrary.org/content/publication/9789264229358-en (cit. on p. 9).

[76] S. O. Ose. “Using Excel and Word to structure qualitative data”. In: *Journal of Applied Social Science* 10.2 (2016), pp. 147–162 (cit. on p. 35).

[77] J. Patel. “Bridging data silos using big data integration”. In: *International Journal of Database Management Systems* 11.3 (2019), pp. 01–06 (cit. on p. 10).

[78] J. Pérez, M. Arenas, C. Gutierrez. “Semantics and complexity of SPARQL”. In: *ACM Transactions on Database Systems (TODS)* 34.3 (2009), pp. 1–45 (cit. on p. 25).

[79] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, D. Vrgoč. “Foundations of JSON Schema”. In: *Proceedings of the 25th International Conference on World Wide Web*. WWW ’16. Montréal, Québec, Canada: International World Wide Web Conferences Steering Committee, 2016, pp. 263–273. ISBN: 9781450341431. DOI: 10.1145/2872427.2883029. URL: https://doi.org/10.1145/2872427.2883029 (cit. on pp. 10, 13).

[80] D. Pilone, N. Pitman. *UML 2.0 in a Nutshell*. Ö’Reilly Media, Inc.”, 2005 (cit. on p. 19).

[81] J. Pleiss. “Standardized data, scalable documentation, sustainable storage–EnzymeML as a basis for FAIR data management in biocatalysis”. In: *ChemCatChem* 13.18 (2021), pp. 3909–3913 (cit. on p. 28).

[82] J. Range, C. Halupczok, J. Lohmann, N. Swainston, C. Kettner, F. T. Bergmann, A. Weidemann, U. Wittig, S. Schnell, J. Pleiss. “EnzymeML—a data exchange format for biocatalysis and enzymology”. In: *The FEBS Journal* 289.19 (2022), pp. 5864–5874 (cit. on p. 28).

[83] J. Raubenheimer. “Excel-lence in data visualization?: the use of Microsoft Excel for data visualization and the analysis of big data”. In: *Data visualization and statistical literacy for open and big data*. IGI Global, 2017, pp. 153–193 (cit. on p. 35).

[84] J. Rumbaugh, I. Jacobson, G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN: 0321245628 (cit. on p. 19).

[85] G. Sander. “Layout of compound directed graphs”. In: (1996) (cit. on p. 21).

[86] E. E. Schadt, M. D. Linderman, J. Sorenson, L. Lee, G. P. Nolan. “Computational solutions to large-scale data management and analysis”. In: *Nature reviews genetics* 11.9 (2010), pp. 647–657 (cit. on p. 9).

[87] M. Schordan. “The language of the visitor design pattern.” In: *J. Univers. Comput. Sci.* 12.7 (2006), pp. 849–867 (cit. on p. 59).

[88] G. Schreiber, F. Gandon. *RDF 1.1 XML Syntax*. W3C Recommendation. https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/. W3C, Feb. 2014 (cit. on p. 23).

[89] A. Seaborne, G. Carothers. *RDF 1.1 TriG*. W3C Recommendation. https://www.w3.org/TR/2014/REC-trig-20140225/. W3C, Feb. 2014 (cit. on p. 23).

[90] I. C. Siffa, J. Schäfer, M. M. Becker. “Adamant: a JSON schema-based metadata editor for research data management workflows”. In: *F1000Research* 11 (2022) (cit. on pp. 15, 32).

[91] I. C. S. Silva, G. Santucci, C. M. D. S. Freitas. “Visualization and analysis of schema and instances of ontologies for improving user tasks and knowledge discovery”. In: *Journal of Computer Languages* 51 (2019), pp. 28–47 (cit. on p. 27).

[92] I.-Y. Song, M. Evans, E. K. Park. “A comparative analysis of entity-relationship diagrams”. In: *Journal of Computer and Software Engineering* 3.4 (1995), pp. 427–459 (cit. on p. 18).

[93] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, N. Lindström. “JSON-LD 1.1”. In: *W3C Recommendation, Jul* (2020) (cit. on p. 23).

[94] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, N. Lindström. “JSON-LD 1.1”. In: *W3C Recommendation, Jul* (2020) (cit. on p. 27).

[95] C. Steinbeck, O. Koepler, F. Bach, S. Herres-Pawlis, N. Jung, J. Liermann, S. Neumann, M. Razum, C. Baldauf, F. Biedermann, et al. “NFDI4Chem-towards a national research data infrastructure for chemistry in Germany”. In: *Research ideas and outcomes* 6 (2020), e55852 (cit. on p. 28).

[96] P. Strömert, J. Hunold, A. Castro, S. Neumann, O. Koepler. “Ontologies4Chem: the landscape of ontologies in chemistry”. In: *Pure and Applied Chemistry* 94.6 (2022), pp. 605–622 (cit. on p. 28).

[97] N. Swainston, A. Baici, B. M. Bakker, A. Cornish-Bowden, P. F. Fitzpatrick, P. Halling, T. S. Leyh, C. O’Donovan, F. M. Raushel, U. Reschel, et al. “STRENDA DB: enabling the validation and sharing of enzyme kinetics data”. In: *The FEBS journal* 285.12 (2018), p. 2193 (cit. on p. 28).

[98] T. J. Teorey, D. Yang, J. P. Fry. “A logical design methodology for relational databases using the extended entity-relationship model”. In: *ACM Computing Surveys (CSUR)* 18.2 (1986), pp. 197–222 (cit. on p. 18).

[99] A. Vaswani. “Attention is all you need”. In: *Advances in Neural Information Processing Systems* (2017) (cit. on p. 74).

[100] R. Y. Wang, D. M. Strong. “Beyond accuracy: What data quality means to data consumers”. In: *Journal of management information systems* 12.4 (1996), pp. 5–33 (cit. on p. 10).

[101] D. S. Wile. “Abstract syntax from concrete syntax”. In: *Proceedings of the 19th international conference on Software engineering*. 1997, pp. 472–480 (cit. on p. 59).

[102] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne, et al. “The FAIR Guiding Principles for scientific data management and stewardship”. In: *Scientific data* 3.1 (2016), pp. 1–9 (cit. on p. 28).

[103] F. Yergeau, T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler. “Extensible markup language (xml) 1.0”. In: *XML*. Vol. 1. 1. Citeseer. 2006, W3C (cit. on p. 9).

[104] F. Zaeri, J. O. B. Rotimi, M. R. Hosseini, J. Cox. “Implementation of the LPS using an excel spreadsheet: A case study from the New Zealand construction industry”. In: *Construction Innovation* 17.3 (2017), pp. 324–339 (cit. on p. 35).

All links were last followed on September 30 2024.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature