MetaConfigurator: A User-Friendly Tool for Editing Structured Data Files

Felix Neubauer[1], Paul Bredl[1], Minye Xu[1], Keyuriben Patel[1], Jürgen Pleiss[2], Benjamin Uekermann[1]

[1]Institute for Parallel and Distributed Systems, University of Stuttgart, Universitätsstraße 38, Stuttgart, 70569, Germany.
[2]Institute of Biochemistry and Technical Biochemistry, University of Stuttgart, Allmandring 31, Stuttgart, 70569, Germany.

Contributing authors: felix@neuby.de;

Abstract

Textual formats to structure data, such as JSON, XML, and YAML, are widely used for structuring data in various domains, from configuration files to research data. However, manually editing data in these formats can be complex and time-consuming. Graphical user interfaces (GUIs) can significantly reduce manual efforts and assist the user in editing the files, but developing a file-format-specific GUI requires substantial development and maintenance efforts. To address this challenge, we introduce *MetaConfigurator*: an open-source web application that generates its GUI depending on a given schema. Our approach differs from other schema-to-UI approaches in three key ways: 1) It offers a unified view that combines the benefits of both GUIs and text editors, 2) it enables schema editing within the same tool, and 3) it supports advanced schema features, including conditions and constraints. In this paper, we discuss the design and implementation of *MetaConfigurator*, backed by insights from a small-scale qualitative user study. The results indicate the effectiveness of our approach in retrieving information from data and schemas and in editing them.

Keywords: rdm, json, yaml, schema, editor, configuration, data, research data management, tool, gui

1 Introduction

Textual formats to structure data, such as JSON, XML, and YAML, are often used for configuration files or to structure research data since they can be read and maintained by humans, as well as deserialized and used by computer programs. The format of data structures can be defined by so-called schemas, which define the rules the data has to conform to. Given a schema, it can be validated whether a particular file confirms to that schema. We refer to all file instances using such formats as *structured data files*. Depending on the domain of such structured data files, they can be complex and time-consuming to modify and maintain when working only with a text editor. Tooling, such as graphical user interfaces, can significantly reduce manual efforts and assist the user in editing the files. Those graphical user interfaces (GUIs), however, require initial effort to be developed, as well as continuous effort in being maintained and updated when the underlying data schema changes. We tackle this problem by developing a web application that automatically generates such assisting GUIs, based on the given data schema. Our approach differs from other schema-to-UI approaches in the following:

1. The tool combines the assistance of a GUI with the flexibility and speed of a text editor by providing both in one view.
2. The schema can be edited using the same tool and type of view.
3. We support more complex schema features, such as conditions and constraints.

We call the tool *MetaConfigurator*. As of writing this paper, it supports JSON and YAML files and uses JSON schema as schema language.

The remainder of this paper is structured as follows: In section 2, we discuss related work and existing schema formats, as well as schema-to-gui approaches. Section 3 describes the design and implementation of *MetaConfigurator*. This includes a description of the schema preprocessing steps, which are necessary to support some advanced schema features. To gather feedback and verify whether the tool can and will be used in the real world, we conducted a user study, which is described in section 4. We discuss the implications of the results in section 5. Finally, we conclude our work in section 6.

2 Background and Related Work

This section covers existing approaches to generate UIs from schemas and other related work.

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 [1]. Because of the popularity of JSON, there is also a demand for a schema language for JSON. One such language is JSON schema [2, 3].

JSON schema has evolved to being the de-facto standard schema language for JSON documents [4]. Schemas for many popular structured data file types exist. *JSON schema store*[1] 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.

[1] https://www.schemastore.org/json/, accessed 2024/01/22

We remark that JSON schema and other schema languages for JSON can also be applied to YAML as JSON is a subset of YAML, although some syntactical details of YAML cannot be expressed with JSON schemas. Thus, we can use JSON schema as a schema language for both JSON and YAML files.

2.2 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 [5, 6, 7] or entity-relationship diagrams [8]. There exist various approaches that generate web forms from the more modern format, JSON schema, e.g., *React JSON Schema Form*[2], *Angular Schema Form*[3], *Vue Form Generator*[4], *JSON Forms*[5], *JSON Editor*[6], and *JSON Form*[7].

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. Those techniques, however, only provide the GUI for editing the data, but not a text-based editor. A text-based editor is useful, especially for experienced users who prefer to edit the data directly. Also, these techniques do not provide a way to edit the schema itself, but only the data. Some of the listed approaches also require an additional *UI schema* to configure the generated form. While these UI schemas allow more customization of the generated form, they also need to be created and maintained. We aim to avoid the need for additional configuration and provide a unified view for editing both the schema and the data.

[2] https://github.com/rjsf-team/react-jsonschema-form, accessed 2024/01/22
[3] https://github.com/json-schema-form/angular-schema-form, accessed 2024/01/22
[4] https://github.com/vue-generators/vue-form-generator, accessed 2024/01/22
[5] https://jsonforms.io, accessed 2024/01/22
[6] https://jsoneditoronline.org, accessed 2024/01/22
[7] https://github.com/jsonform, accessed 2024/01/22

2.3 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[8] 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*[9], *Liquid Studio*[10], *XML ValidatorBuddy*[11], *JSON-Buddy*[12], *XMLBlueprint*[13], and *Oxygen XML Editor*[14]. 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.

2.4 Adamant

Adamant[15] is a JSON Schema-based form generator and schema editor specifically designed for scientific data [9]. 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.

[8] https://jsoneditoronline.org, accessed 2024/01/22

[9] https://www.altova.com/xmlspy-xml-editor, accessed 2024/01/22

[10] https://www.liquid-technologies.com/json-schema-editor, accessed 2024/01/22

[11] https://www.xml-buddy.com/, accessed 2024/01/22

[12] https://www.json-buddy.com/, accessed 2024/01/22

[13] https://www.xmlblueprint.com, accessed 2024/01/22

[14] https://www.oxygenxml.com, accessed 2024/01/22

[15] Current version of Adamant as of writing this paper: Adamant v1.2.0

Besides the frontend, Adamant also provides a backend that allows the integration into other tools as well as storing and reusing schemas. Adamants 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].

2.5 Schema Visualization

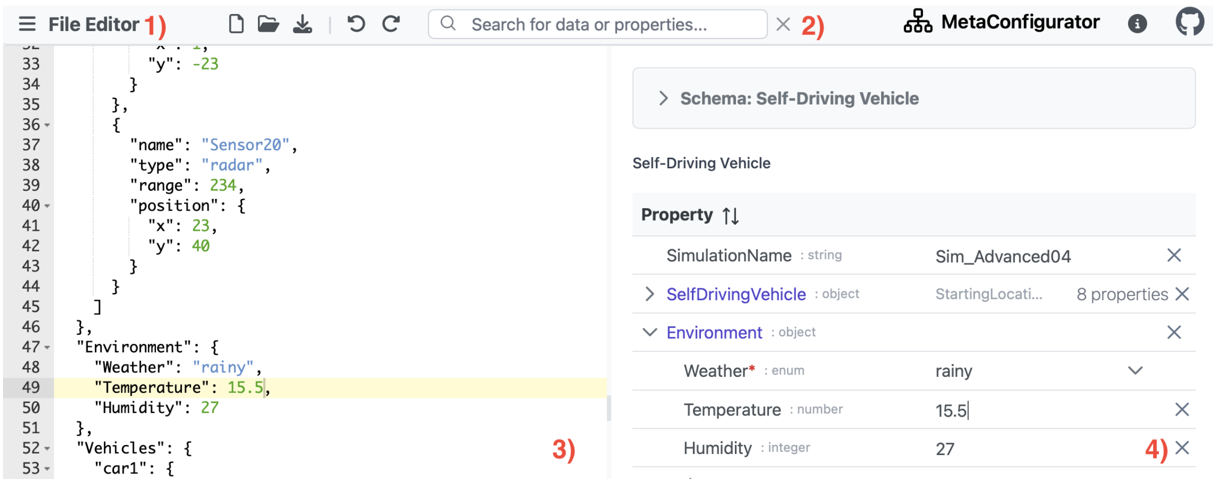
Generating a GUI from a schema is related to schema visualization, for which several techniques exist [10, 11, 12, 13]. However, the focus of schema visualization is on providing a static visual representation of the schema and not on providing a GUI for editing the schema. Thus, we do not consider schema visualization approaches in this work.

3 Implementation

We propose a novel approach that differs from existing work: *MetaConfigurator* 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). In addition to the UI for editing the data, we also propose a schema editor that shares the same UI and inherently supports all JSON schema keywords. This is achieved by treating the schema itself as a *structured data file* and generating the schema editor GUI from a JSON schema meta-schema. This section discusses the design and implementation of our tool. *MetaConfigurator* is a client-side web application, which means that it runs in the browser of the user and does not require a server. It uses the Vue.js framework[16]. The code is open-source and available on GitHub[17].

[16] https://github.com/vue-generators/vue-form-generator, accessed 2024/01/22

[17] https://github.com/PaulBredl/meta-configurator/, accessed 2024/01/22



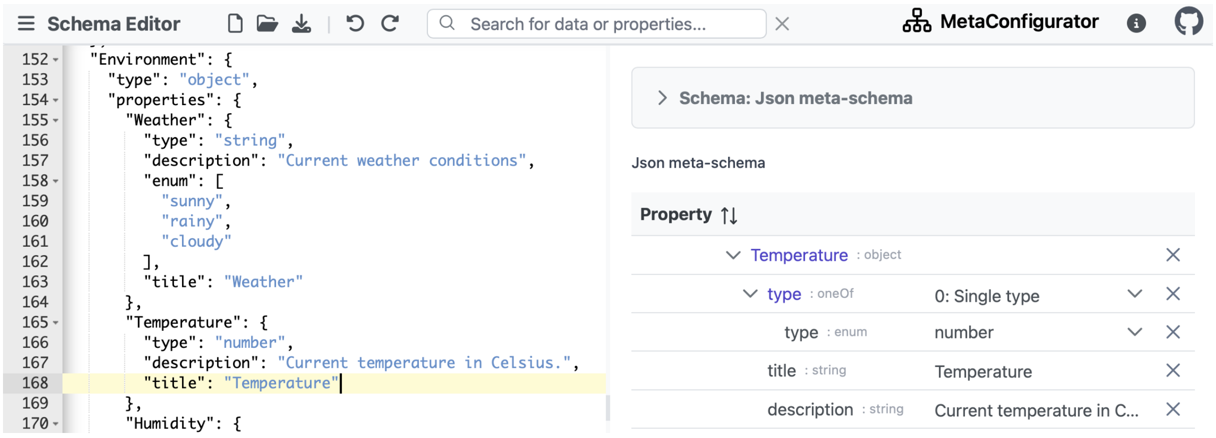
```
          "y": -23
        }
      },
      {
        "name": "Sensor20",
        "type": "radar",
        "range": 234,
        "position": {
          "x": 23,
          "y": 40
        }
      }
    ]
  },
  "Environment": {
    "Weather": "rainy",
    "Temperature": 15.5,
    "Humidity": 27
  },
  "Vehicles": {
    "car1": {
```

Schema: Self-Driving Vehicle

Self-Driving Vehicle

Property	
SimulationName : string	Sim_Advanced04
SelfDrivingVehicle : object	StartingLocati... 8 properties
Environment : object	
Weather* : enum	rainy
Temperature : number	15.5
Humidity : integer	27

Fig. 1 UI of 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



```
"Environment": {
  "type": "object",
  "properties": {
    "Weather": {
      "type": "string",
      "description": "Current weather conditions",
      "enum": [
        "sunny",
        "rainy",
        "cloudy"
      ],
      "title": "Weather"
    },
    "Temperature": {
      "type": "number",
      "description": "Current temperature in Celsius.",
      "title": "Temperature"
    },
    "Humidity": {
```

Schema: Json meta-schema

Json meta-schema

Property	
Temperature : object	
type : oneOf	0: Single type
type : enum	number
title : string	Temperature
description : string	Current temperature in C...

Fig. 2 The schema editor view. The schema editor is based on the same UI as the file editor.

3.1 User Interface

MetaConfigurator aims to make editing structured data files user-friendly and efficient. The UI is inspired by a tool by one of the authors[18]. It has three distinct views:

1. File editor (figure 1): In this view, the user can modify their structured data file, based on a schema.
2. Schema editor (figure 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

[18]https://github.com/Logende/BossShopProEditor, accessed 2024/04/01

Table 1 File data and schema for the different views

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

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 can be treated as such, and the tool can offer assistance accordingly. The user can edit the schema in the same way as they can edit their data. We use an adapted version of the JSON schema meta-schema 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. We even treat the settings of the tool as a structured data file, for which we define an underlying settings schema and use the tool itself to edit the settings. Table 1 shows which data and schema the tool uses in the different views.

3.2 Architecture

The core of *MetaConfigurator* is a single source of truth data store that contains the current user configuration data (as a JavaScript Object). With this data store, we can bidirectionally connect what we call *editor panels*. An editor panel is a modular component the user can access to modify the data indirectly. We implement two editor panels: a text editor and a graphical user interface, but the approach can be extended to any other way in which the data can be presented, e.g., a graph view of the data. Every editor panel subscribes to the changes in the data store, so it is updated accordingly whenever the data changes. Additionally, every panel has the capability of updating the data store themselves.

3.3 Text Editor

For the text editor, we use the *Ace Editor*[19] library to embed an interactive text editor into our user interface. It provides useful features for our approach, such as syntax highlighting and code folding. To provide the user feedback on whether their data is valid according to the provided schema, we perform schema validation. We make use of the *Ajv JSON schema validator*[20] library. If schema violations are found, the corresponding lines in the text editor will be marked with a red error hint, which also describes the violation.

The panel subscribes to the data store. Whenever the configuration data is changed in the store, the panel will take the new configuration data JavaScript Object, serialize it into the given data format, and replace the text in the text editor with the new serialized data. The action of replacing the text in the text editor will cause formatting and comments to be lost, which we accept. In the future, mechanisms can be applied to avoid the loss of formatting or comments. When the user edits the text in the text editor, the text is deserialized into a JavaScript Object and sent to the data store, which then updates the configuration data object and notifies all other subscribed panels of the change. The tool can support any data format if the following functionality is implemented:

1. Parsing string in the data format as a JavaScript object (mandatory)
2. Stringify JavaScript object into a string of the data format (mandatory)
3. Function to determine line within the text editor, based on a given data path
4. Function to determine the current selected data entry, based on the configuration text and a cursor position

The last two functionalities are optional but can significantly improve the user experience. They allow the tool to synchronize the cursor position in the text editor with the selected data entry in the GUI editor and vice versa. We have implemented all functionality for JSON and *1.* and *2.* for YAML. To tackle *3.* and *4.*, the tool parses the JSON string into a concrete syntax tree and

[19] https://ace.c9.io/, accessed 2024/01/22
[20] https://ajv.js.org, accessed 2024/01/22

Table 2 Corresponding GUI implementations for JSON schema types/keywords

Type/keyword	**GUI implementation**
`string`	Text field
`number`	Text field for floating point numbers with increment and decrement button
`integer`	Text field for integer numbers with increment and decrement button
`boolean`	`true`/`false` toggle
`object`	Expandable list of child columns in the properties table
`array`	Similar to objects, but also has a button to add new items
`enum`	Dropdown menu
`const`	Dropdown menu with just one entry
`required`	Red asterisk to the left of of the property name
`deprecated`	Strikethrough styling for property name
`anyOf`	Multiselect menu to choose sub-schemas. Based on the selected sub-schemas, corresponding properties will be shown as children in the table.
`oneOf`	Dropdown menu to choose one sub-schema. Based on the selected sub-schema, corresponding properties will be shown as children in the table.

traverses the tree to determine the connection between data objects and their location within the text. The text editor has more functionalities, such as the possibility to open a file by drag-and-drop into the editor, undo/redo operations, and the possibility to change the font size.

3.4 GUI Editor

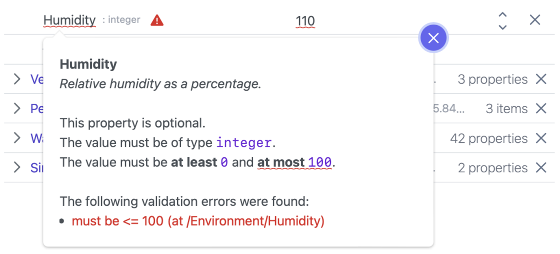
The GUI Editor is automatically generated based on the JSON schema that the user provides. It is structured in a table-like way, where each row represents a key-value pair of the configuration data. Array elements are represented similarly, where the index of the array element is the key and the value is the array element itself. Figure 1 shows the GUI editor component with an example schema and configuration data.

To allow this representation of the schema, we do some preprocessing of the schema, which is described in section 3.5. To assist the user in editing the configuration data, the GUI editor offers a set of features:

- *Traversal of the Data Tree*: By default, only the first level of the data tree is shown. The user can expand the data tree by clicking on the arrow next to the key of an object or array to show sub-properties or array elements. To keep the interface clear, the user can also click on the property name or array index to *zoom in* on that element. This will show the sub-properties of that element at the top level as if that property was the root of the data tree. The breadcrumb at the top allows the user to see on which path the GUI editor is currently zoomed in and to navigate back to the upper levels of the tree.
- *GUI Implementation for JSON Schema Features*: Table 2 provides an overview of how our approach handles the different JSON schema keywords and types. We support most of the JSON schema keywords and types, especially the ones that are most commonly used[21].
- *Remove Data*: The user can delete properties or array elements from the data by clicking on the × button next to the edit field. This button is only visible if the property is not required and there exists data.
- *Schema Information Tooltip*: When the user hovers over a property, an overlay is displayed (figure 3), showing information from the schema, such as title, description, constraints (e.g., the minimum and maximum value for numbers) and schema violations if there are any. This feature helps the user to understand the constraints and the meaning of a property.
- *Highlighting Schema Validation Errors*: When the configuration data does not comply with the schema, the corresponding

[21]In the GitHub wiki (https://github.com/PaulBredl/meta-configurator/wiki/JSON-schema-keyword-support), we provide a detailed overview of which JSON schema features are supported by the tool

elements are underlined in red and highlighted with a red error icon. This way, the user knows what parts of the data are invalid and what the error is.


Humidity : integer ⚠ 110

Humidity
Relative humidity as a percentage.

This property is optional.
The value must be of type `integer`.
The value must be **at least** `0` and **at most** `100`.

The following validation errors were found:
- must be <= 100 (at /Environment/Humidity)

Fig. 3 Tooltip with a schema violation

3.5 Schema Preprocessing

To represent the schema in the GUI editor, it is necessary to preprocess the schema as for some JSON schema keywords it is not immediately clear how to represent them in a GUI. For example, the `type` keyword can have multiple values, which represent a type union. There is no obvious GUI component that represents this case. We differentiate between three ways of preprocessing: A one-time preprocessing step when loading the schema, an internal preprocessing that happens at every layer of the schema tree, and calculating an effective schema that happens every time the configuration data changes. It is important to note that all these preprocessing steps are only used for generating the GUI editor. They will not affect the schema file itself, so the user will always see the original schema file in the schema editor. The following sections describe the preprocessing steps in detail and how we solve cases such as the one just mentioned.

3.5.1 One-time Preprocessing Step

When the schema is loaded, we perform a one-time preprocessing step. This step only processes the whole schema once and does not depend on the configuration data. We do not perform any time-consuming operations in this step, so the user does not have to wait for the GUI to load. The following three steps are performed in this preprocessing step:

1. Title inducing: If a property does not have the keyword `title`, we inject the property name as the `title`. The `title` can be then used in various places in the GUI, such as the tooltip as a placeholder in the text field.
2. Processing `enum` and `const`: The `const` keyword is used to restrict the value of a property to a single value. It is semantically equivalent to the `enum` keyword with a single value. Thus, we convert any usage of `const` to an `enum` with a single element, which allows us to ignore the `const` keyword in other operations.
3. Inferring types of enums: If the `enum` keyword is used, but not the `type` keyword, we infer the type of the property from the elements of the `enum` array. This is useful for the GUI editor as it allows us to show the correct type information in the tooltip.

3.5.2 Lazy Preprocessing

The following preprocessing steps happen at every layer of the schema tree lazily, only when the user expands the corresponding property in the GUI editor. Laziness of the preprocessing is required as schemas can have circular references, which would, otherwise, lead to infinite loops. The following steps are executed in this preprocessing step:

1. Resolving references (We currently only support references to schemas in the same file).
2. Merging `allOf` sub-schemas into the parent schema.
3. Removing `oneOf` and `anyOf` sub-schemas that can never be satisfied in combination with the parent schema.

3.5.3 Calculating the Effective Schema

This third preprocessing step is calculated every time the data changes. The JSON schema keywords `if`, `then`, and `else` provide a way to include conditions in the JSON schema. If the schema in the `if` field is valid, then also the schema in the `then` field must be valid, otherwise, the schema in the `else` field must be valid. This makes the schema data dependent. To show the correct properties, the tool evaluates the data and dependent on validity or not, either uses the `then` or the `else` schema. The approach similarly handles the `dependentRequired` and the `dependentSchemas` keywords. For schemas without any of those keywords, this step is trivial as the schema is not modified in any way.

3.6 Developing a Custom Meta-Schema

The schema editor view, mirroring the file editor structure, employs the JSON schema meta-schema instead of the user-provided schema to generate the GUI panel. However, applying our approach to the official JSON schema meta-schema [3] yields a non -intuitive editor. We address this by developing a new meta-schema that overcomes the shortcomings of the official one.

3.6.1 Missing Descriptions

The `description` keyword allows schema authors to provide context to schema elements, aiding users in understanding the meaning of properties. Since the JSON schema meta-schema lacks descriptions, we augment our modified meta-schema with descriptions from the JSON schema specification [3].

3.6.2 External References

MetaConfigurator currently lacks support for external schema references and the `$vocabulary` keyword used in the JSON schema meta-schema. To overcome this limitation, we consolidate all schemas into a single schema file within the `$defs` object and replace external references with local ones.

3.6.3 Dynamic Anchors and References

The JSON schema meta-schema employs dynamic references and anchors to dynamically extend the schema at runtime. To simplify and enhance compatibility, we replace these dynamic features with non-dynamic references using the `$ref` keyword.

3.6.4 Restricting Keywords

The JSON schema meta-schema permits all fields in any context, leading to overwhelming displays of irrelevant fields in certain contexts. To address this, we refine the meta-schema to restrict the presentation of fields based on context, addressing feedback from our user study.

Thus, we added `if` conditions to each field to only show them when they make sense in the current context. For example, the relevant properties for arrays are only visible when the current property is of type array.

3.6.5 Hiding Advanced Fields

To further reduce the number of fields shown to the user, we introduce a mechanism to hide fields that are not necessary for the basic usage of the schema. Baazizi et al. [4] analyzed the usage of JSON schema keywords in 82,000 JSON schemas. Based on their findings, we put all less common fields in an *advanced* section of the GUI. This is implemented by introducing a custom keyword `advanced` to our meta-schema. This keyword is a boolean field, which is `false` by default.

4 User Study

During development, we conducted a user study with five participants to ensure a user-centered process and identify areas for improvement in our tool. 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.

During each interview, we shortly introduced *MetaConfigurator*, gave the participant tasks to execute using the tool (retrieving data from a file, modifying a file, and editing the schema), and finished the session with open-ended questions on how the tool could be improved. The interview tasks and additional resources regarding the user study can be found on GitHub[22]. The responses of the participants helped us identify 23 aspects of the tool that can be improved, of which we have addressed 20, while 3 remain for future work.

For example, the aspect *validation errors should be highlighted in a stronger way* was addressed by introducing a red error icon and underlining the invalid elements in red. 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

[22] https://github.com/PaulBredl/meta-configurator/tree/main/paper/user_study

noted that they can imagine using *MetaConfigurator* in practice for their own work. Three participants highlighted the *intuitiveness* of our 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. Although the user study's small sample size of five participants limits the generalizability of the results, the successful completion of all tasks and the feedback suggest the tool's practical applicability and user-friendliness, with minor exceptions in the schema editor.

5 Discussion

This section discusses the limits and implications of our work as well as possible future work.

5.1 Implications of Our Work

MetaConfigurator provides a novel approach for manually editing structured data files and thus has several implications. First, it can help to improve data consistency, which is important for many applications, including scientific applications, as the approach helps to avoid errors. Second, it can enhance data accessibility, as it allows users to edit structured data files without requiring in-depth knowledge of JSON or YAML syntax or the intricacies of JSON schema semantics. Last, it can foster productivity by reducing the time and effort required to manually edit configuration files, especially for large and complex projects.

5.2 Limitations

JSON schema keywords that are infrequent[4] and more challenging to provide support for (`$anchor`, `$dynamicRef`, `$dynamicAnchor`, `not`), not relevant for editing files (`readOnly`, `writeOnly`), deprecated in JSON schema (`dependencies`, `$recursiveAnchor`, `$recursiveRef`) or highly specific (`prefixItems`, `unevaluatedItems`, `unevaluatedProperties`) are not (yet) supported by the GUI editor of *MetaConfigurator*. The text editor supports all keywords and the schema editor also suggests all keywords to the user when they edit their schema. Further, YAML support is only partially implemented (see section 3.3) and modifying a YAML document from the GUI Editor will remove all comments in the document. Also, the user study revealed that editing schemas with *MetaConfigurator* is not as intuitive as editing data files, especially for users who are not familiar with JSON schema. JSON schema may be feature-rich and expressive, but it is also complex and hard to understand for new users. Therefore, schema editors that are tailored to a specific schema language may be more suitable for creating and editing schemas.

5.3 Future Work

Possible next milestones for *MetaConfigurator* are: Existing limitations can be addressed, for example, by implementing full YAML support or making the schema editor more user friendly. The application can be extended to support more data formats, such as XML, and other schema languages, such as XML schema[23]. Also, multiple users of the tool requested support for ontologies, which could be achieved by supporting ontology languages such as OWL[24] or a linked data format like JSON-LD[25].

Evaluating *MetaConfigurator's* effectiveness in a real-world scenario is another major milestone. This could involve applying it to a specific use case to assess its practical value. Alternatively, or in conjunction with a real-world application, a user study with a larger and more diverse participant pool could be conducted. This would allow for broader, more generalizable conclusions about the tool's strengths and weaknesses. Alongside these milestones, the tool is planned to be further developed and maintained.

6 Conclusion

In conclusion, this paper presents *MetaConfigurator*, a novel tool designed to simplify the process of managing and modifying structured data files. Our approach allows users to edit structured data files in a GUI, while still having the flexibility and speed of a text editor. Additionally, it removes

[23] https://www.w3.org/XML/Schema, accessed 2024/04/03
[24] https://www.w3.org/TR/owl2-overview/, accessed 2024/04/03
[25] https://json-ld.org/, accessed 2024/04/03

the need for developing and maintaining a custom GUI for a particular schema. We use JSON schema as a schema language for the tool as it is an expressive and popular schema language, and this paper elaborates on the methodologies we use to generate a user interface from a JSON schema. The user study conducted during the development of *MetaConfigurator* suggests that the tool is user-friendly and holds promise for real-world application. The tool's implications extend to enhancing data consistency, data accessibility, and productivity.

Statements and Declarations

We thank the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) for supporting this work by funding – EXC2075 – 390740016 under Germany's Excellence Strategy. Furthermore, we acknowledge the support of the Stuttgart Center for Simulation Science (SimTech). Last, we want to thank all participants of the user study and also Jan Range for the fruitful discussions.

Author contributions statement

- **Conceptualization**: all authors;
- **Writing – original draft preparation**: Neubauer, Felix; Bredl, Paul;
- **Writing – review and editing**: Uekermann, Benjamin; Pleiss, Jürgen;
- **Supervision**: Uekermann, Benjamin; Pleiss, Jürgen.
- **Software**: Neubauer, Felix; Bredl, Paul; Xu, Minye; Patel, Keyuriben.

References

[1] Marrs, T. *JSON at work: practical data integration for the web* (O'Reilly Media, Inc., 2017).

[2] Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M. & Vrgoč, D. *Foundations of json schema*, ACM Digital Library, 263–273 (International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 2016). URL https://doi.org/10.1145/2872427.2883029.

[3] JSON Schema — json-schema.org. https://json-schema.org. [Accessed 01-05-2023].

[4] Baazizi, M.-A., Colazzo, D., Ghelli, G., Sartiani, C. & Scherzinger, S. An Empirical Study on the "Usage of Not" in Real-World JSON Schema Documents (Long Version) (2021). URL https://arxiv.org/abs/2107.08677.

[5] Kasarda, J., Nečaský, M. & Bartoš, T. *Generating XForms from an XML Schema*, 706–714 (Springer Berlin Heidelberg, 2010). URL http://dx.doi.org/10.1007/978-3-642-14306-9_70.

[6] Fenech, J. *A semantic editor: generation of a web-based user interface from an XML schema definition.* B.S. thesis, University of Malta (2008).

[7] Kuo, Y. S., Shih, N. C., Tseng, L. & Hu, H.-C. *Generating form-based user interfaces for xml vocabularies*, DocEng05 (ACM, 2005). URL http://dx.doi.org/10.1145/1096601.1096619.

[8] Bajaj, A. & Knight, J. *User Interface Generation from the Data Schema*, 145–153 (IGI Global). URL http://dx.doi.org/10.4018/978-1-60566-344-9.ch011.

[9] Siffa, I. C., Schäfer, J. & Becker, M. M. Adamant: a JSON schema-based metadata editor for research data management workflows. *F1000Research* **11**, 475 (2022). URL https://doi.org/10.12688/f1000research.110875.2.

[10] Frasincar, F., Telea, A. & Houben, G.-J. *Adapting Graph Visualization Techniques for the Visualization of RDF Data*, 154–171 (Springer London, London, 2006), second edition edn. URL https://doi.org/10.1007/1-84628-290-X_9.

[11] Silva, I. C. S., Santucci, G. & Freitas, C. M. D. S. Visualization and analysis of schema and instances of ontologies for improving user tasks and knowledge discovery. *Journal of Computer Languages* **51**, 28–47 (2019). URL https://www.sciencedirect.com/science/article/pii/S1045926X17302458.

[12] Deligiannidis, L., Kochut, K. J. & Sheth, A. P. *RDF Data Exploration and Visualization*, CIMS '07, 39–46 (Association for Computing Machinery, New York, NY, USA, 2007). URL https://doi.org/10.1145/1317353.1317362. Title from The ACM Digital Library.

[13] North, C., Conklin, N. & Saini, V. *Visualization schemas for flexible information visualization*, INFVIS-02, 15–22 (IEEE Comput. Soc, 2002). URL https://doi.org/10.1109/infvis.2002.1173142.