# Design, Implementation, and Evaluation of a Meta Configuration Tool Using a Schema-to-UI Approach

Felix Neubauer, Paul Bredl, Minye Xu, Keyuriben Patel

***Abstract*—Textual formats to structure data, such as JSON, XML and YAML, are often used for configuration files or to structure measurement and research data. Depending on the domain and underlying schema, the data can be complex and time-consuming to modify and maintain. Graphical user interfaces (GUIs) have proven effective in simplifying data management but entail substantial development and maintenance efforts. To address this challenge, we introduce a novel approach in this paper: a meta-program that automatically generates GUIs tailored to a given data schema. Our approach differentiates itself from others by 1) offering a unified view that combines the benefits of both GUIs and rich-text code editors, 2) enabling schema editing within the same tool and 3) support of advanced schema features like conditions and constraints. We evaluate various schema formats and select JSON schema as the most suitable for our work. Then we present the program design, implementation, and the results of a user study involving three professors and two researchers. Our approach offers a practical solution that makes data and schema management easier.**

***Index Terms*—JSON, YAML, configuration, schema, gui**

## I. Introduction

Textual formats to structure data, such as JSON, XML and YAML, are human-readable as well as machine-readable. Such formats are often used for configuration files or to structure measurement or research data, since they can be read and maintained by humans, as well as deserialized and used by computer programs. The format of those 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 will call all file instances using such formats *configuration files*. Depending on the domain of such configuration files, they can be complex and time-consuming to modify and maintain. 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 meta program that automatically generates such assisting GUIs, based on the given data schema.

Our approach differs from other schema-to-UI approaches in following:

1) The tool combines the assistance of a GUI with the flexibility and speed of a rich-text code 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.

In section II, we discuss related work and existing schema formats, as well as schema-to-gui approaches. In section III, we evaluate existing schema languages to find the most suitable one for this work. Section IV describes the design and introduces the architecture of *MetaConfigurator*. Next, in section V, we cover the implementation of *MetaConfigurator*. 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 VI. Finally, we conclude our work in section VIII.

## II. Related Work

This section covers existing schema languages and existing approaches to generate UIs from them. As our research is of a practical nature, we also consider gray literature such as specifications of schemas or websites.

### A. Schema Languages

*Schema languages* are formal languages that specify the structure, constraints, and relationships of data, for example in a database or structured data formats.

As this work is concerned with generating a GUI based on a schema, we need to choose a suitable schema language. The following sections describe existing schema languages. We will compare them in section III to determine which is the most suitable one for this work.

*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]. Listing 1 shows an example of a JSON schema and listing 2 shows an example of a JSON document that conforms to the schema.

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* [5] 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. [6], [7] give further examples of JSON schema used in practice.

JSON and YAML documents are of a similar structure (JSON is a subset of YAML) and JSON schema can be applied to YAML documents too. Some syntactical details of YAML can, however, not be expressed with JSON schema.

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

Listing 1: JSON schema example

```json
{
  "firstName": "John",
  "lastName": "Doe",
  "age": 21
}
```

Listing 2: JSON example for the schema in listing 1

*2) XSD and DTD:* For XML the two de-facto standard schema languages are Document Type Definition (DTD) [8] and XML Schema Definition (XSD) [9]. XSD is the newer and more expressive format and in large parts replaces and supersedes the more limited format DTD [10]. It is recommended by W3C as a schema language for XML documents [9]. Multiple other schema languages have been proposed and developed but are relatively unknown compared to XSD [11], [12].

*3) Other schema languages:* We also consider the following schema languages:

(a) CUE (Configure, Unify, Execute) [13] is a data validation and configuration language, which can be used with various data formats, such as JSON and YAML (it is a superset of both). It has several use cases, especially in configuration and data validation.
(b) Apache Avro [14] is an open-source project that provides data serialization and data exchange services for Apache Hadoop. It uses a JSON-based schema language.
(c) JSON Type Definition (JTD) [15] is a schema language for JSON documents, which is significantly simpler than JSON schema.
(d) Type Schema [16] is a schema language for JSON documents, similar to JSON Type Definition but using a different syntax.
(e) GraphQL schema language [17] is a schema language for GraphQL APIs.
(f) Protocol Buffers [18] is a language for data serialization by Google.

We do not consider any graphical modeling languages, such as UML or ER diagrams, as they are not text-based. Although they can be converted to text-based formats, their main purpose is to model data structures and relationships between them. We also do not consider any ontology languages, such as OWL or RDF Schema, as they are not intended for data validation but rather for knowledge representation. Future work could investigate if such languages are also useful for our use case. Finally, we do not consider any programming languages as schema languages. Technically, programming languages can be used to define data structures and constraints, but they are not intended for this purpose, and it would be very challenging to generate a GUI from them.

### B. Existing Approaches

Our work focuses on assisting users in creating and maintaining configuration files so that they are valid and adhere to a predefined schema.

There exist techniques to validate configuration files against a schema [19]–[21]. Usually, schema validation is done only internally, e.g., by web services or libraries. However, there exist also approaches that use the schema to assist the user in creating and maintaining configuration files. IDEs, such as Visual Studio Code or IntelliJ IDEA, can validate configuration files against a schema and provide the user with error messages. Those IDEs provide also other features, such as auto-completion, syntax highlighting, and tooltips. However, they typically do not provide a graphical user interface (GUI) for editing the configuration files based on the schema.

*1) Form generation:* Related to our work are approaches that generate a GUI from a schema. This section covers form generators, i.e., approaches that generate a web form from a schema. Such forms can assist the user in a multitude of ways, such as help by tooltips, auto-completion (Figure 1) and choice selections (Figure 2). By inherently adhering to the schema structure, with such GUIs configuration errors are avoided or at least significantly reduced. Users who are not very familiar with the configuration schema profit most from the GUI assistance, but even experienced users tend to not remember every individual detail of the schema and benefit.

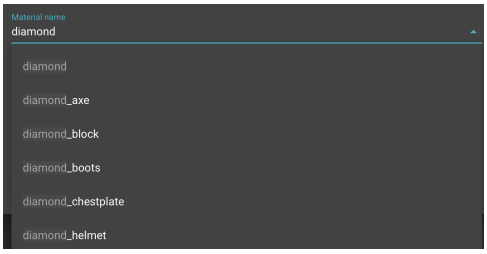


Fig. 1: Auto-Completion

There exist various approaches that generate web forms from a schema, for different frontend frameworks, e.g., *React JSON Schema Form* [22], *Angular Schema Form* [23], *Vue Form Generator* [24], *JSON Forms* [25], *JSON Editor* [26],

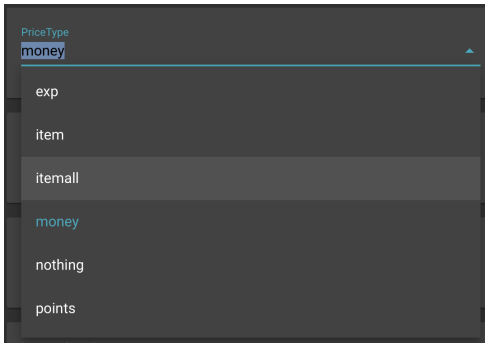


Fig. 2: Choice Selection

and *JSON Form* [27]. Those approaches are all based on JSON schema and generate a form that can be filled out by the user and the resulting JSON document is validated against the schema. If the user enters invalid data, the form shows an error message. The generated forms usually have a specific component for each type of data, e.g. a text field for strings or a number field for numbers, similar to our approach. Figure 3 shows an example of a generated form using JSON Forms.

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. The most significant limitation of all except the last two of the given approaches is that they also require a "UI schema" in addition to the JSON schema, which is used to configure the generated form. While these configurations can be used to customize the generated form, they also need to be created and maintained by the schema author. Consequently, those approaches cannot be used to generate a GUI for any arbitrary schema, but manual effort is required to create the UI schema.

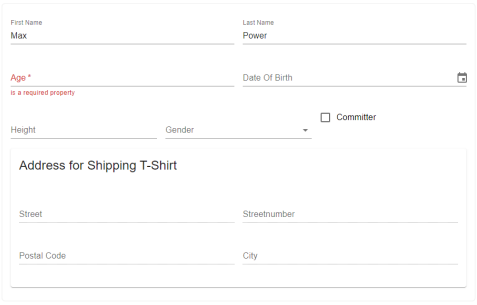


Fig. 3: JSON Forms, example for a generated form

Adamant [28] is a JSON-schema based form generator specifically designed for scientific data. It is similar to our approach in that it generates a GUI from a JSON schema and also allows to edit and create JSON schema documents and differentiates between a schema edit mode and a data edit mode. It supports a subset of JSON schema, which is sufficient for many use cases. In addition to that, it supports the extraction of units from the description of a field, which is helpful for scientific data. Figure 4 shows an example in the schema edit mode. Limitations of Adamant are first, it does only support a subset of JSON schema, which is sufficient for many use cases, but not for any arbitrary schema. Second, it does not provide a text-based editor for neither the schema nor the data. Finally, it is specifically designed for scientific data, which makes it less suitable for other use cases, especially large and complex schemas.

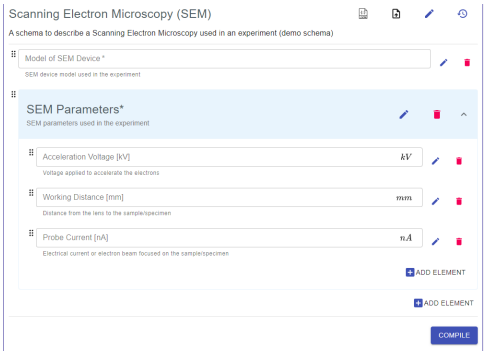


Fig. 4: Adamant, example for a form in edit mode

*2) Schema editors:* In *MetaConfigurator* we aim to provide a GUI for both editing configuration files and editing the schema. For the latter, there exist several so-called schema editors, which are tools for creating and editing schemas that are either text-based or graphical (or both).

*JSON Editor Online* [26] 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 (Figure 5). It provides a text-based or tree-based view for editing the JSON documents. For simple objects that are not further nested, it provides also a table-based view (Figure 6). However, the features of the editor are very 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 the 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* [29], *Liquid Studio* [30], *XML ValidatorBuddy* [31], *JSONBuddy* [32], *XMLBlueprint* [33], and *Oxygen XML Editor* [34]. 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.

*3) Schema visualization:* Generating a GUI from a schema is related to schema visualization, for which several techniques exist [35]–[38]. 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. However, future work could investigate how such techniques could be embedded in our approach.

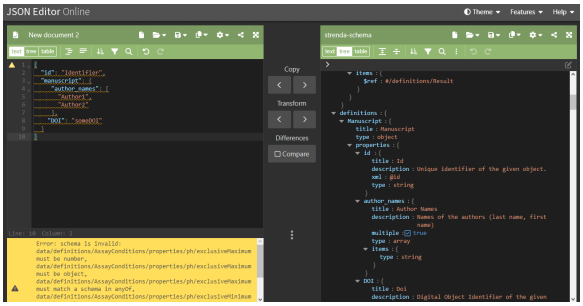


Fig. 5: JSON Editor Online

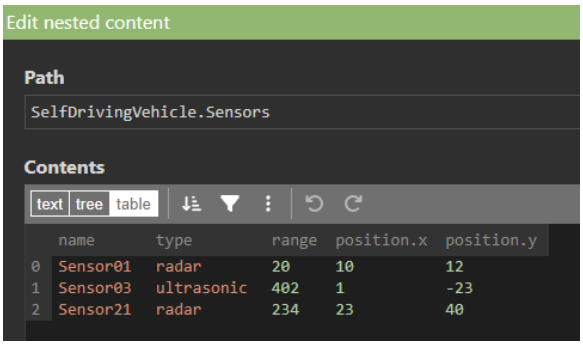


Fig. 6: JSON Editor Online, table view

## III. Evaluation of Schema Languages

We evaluate the schema languages mentioned in section II-A to determine which is the most suitable one for this work.

### A. Evaluation criteria

Ideally, the schema language of *MetaConfigurator* is both popular and supported by numerous tools and libraries as well as expressive enough to express the features we need. We use the following criteria and metrics:

1) **Practical usage** — Ideally our approach uses a schema language that already known by many developers. As indicator of the practical usage we use the approximate search results on stackoverflow.com as metric. We acquire the results by querying the google search engine with the name of the schema language and "site:stackoverflow.com", which limits the search results to stackoverflow.com. This metric might also correlate with the complexity of the schema language as a more complex to use schema language will likely lead to more questions asked on the site. Nevertheless, we assume that a significantly higher number of results indicates that a language is more known than others.
   Additionally, we investigate how well the schema languages are supported by IDEs and code libraries:
   a) *Tool support* — We used the 10 most popular IDEs [39] and checked if the IDE supports the schema language either natively or by a plugin. Support here means that either the IDE is capable of validating documents against a schema in the schema language or supports creating schema files, e.g., by using syntax highlighting for the schema language.
   b) *Library support* — As we implement a web-based tool, we JavaScript or TypeScript bases tools are helpful for our approach, e.g., so we can reuse a package for schema validation. We investigate the number of node modules exist that are related to the schema languages by querying the node module search on www.npmjs.com with the name of the schema language.
2) **Expressiveness** — We evaluate how expressive each of the schema languages are, i.e., what possible constructs the language is able to express. We define eight requirements on the language features that we consider helpful for our approach. The number of requirements a schema language fulfills is our metric that indicates how expressive the language is. Table II reports the results. The nine requirements are:
   a) *Simple types* — This is fulfilled if the schema language provides the possibility to define simple data types, at least strings, numeric types, and a boolean type. This is a fundamental feature for our approach.
   b) *Complex types* — This is fulfilled if the schema language provides the possibility to define complex data types, at least records and arrays. This is crucial feature for our approach as configuration files are often structured data rather than plain key-value pairs.
   c) *Descriptions* — This is fulfilled if the schema language provides the possibility to add descriptions to fields. This is helpful in a schema-to-GUI approach as the description can be shown to the user, providing potential helpful information on how a field should be filled.
   d) *Examples* — This is fulfilled if the schema language provides the possibility to add example values. This is helpful in our approach as the example values can serve as placeholders in the GUI editor.
   e) *Default values* — This is fulfilled if the schema language provides the possibility to add default values which are assumed in an absence of a value. This often helpful information can be displayed to the user or used as placeholder values.
   f) *Optional values* — This is fulfilled if the schema language provides the possibility to declare values as optional or required. Often it is not necessary to provide all values in a configuration file, so it is helpful to mark fields as required or optional in the GUI editor.
   g) *Constraints* — This is fulfilled if the schema language provides the possibility to constrain values of fields, e.g., maximum length of strings. To be exact, for this evaluation we required that at least two of the following constrained can be expressed by the schema language:
      - The length of strings can be limited.
      - The range of numeric types can be limited, e.g., to only positive values.
      - The valid values of a field can be restricted to a finite amount of values (enumeration).
      - The format of a string field can be constrained to a certain pattern.

TABLE I
EVALUATION OF DIFFERENT SCHEMA LANGUAGES

| **Schema language** | **# Search results** | **IDE support** | **# Node packages** | **Expressiveness** |
|---|---|---|---|---|
| JSON schema | 245.000 | 8 / 10 | 4.536 | 9 /9 |
| XSD | 151.000 | 8 / 10 | 116 | 8 / 9 |
| DTD | 69.700 | 9 / 10 | 34 | 6 / 9 |
| CUE | 10.500 | 4 / 10 | 97 | 8 / 9 |
| Avro | 20.000 | 8 / 10 | 211 | 5 / 9 |
| JSON Type Definition (JTD) | 109 | 0 / 10 | 17 | 5 / 9 |
| TypeSchema | 8.450 | 0 / 10 | 5 | 8 / 9 |
| protobuf | 44.800 | 9 / 10 | 1.210 | 4 / 9 |
| GraphQl schema | 31.000 | 7 / 10 | 1.509 | 6 / 9 |

TABLE II
COMPARISON OF EXPRESSIVENESS OF DIFFERENT SCHEMA LANGUAGES

| **Schema language** | Simple types | Complex types | Descriptions | Example values | Default values | Optional values | Constraints | Conditions | References | Result |
|---|---|---|---|---|---|---|---|---|---|---|
| JSON schema | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 9 / 9 |
| XSD | ✓ | ✓ | ✓ | x | ✓ | ✓ | ✓ | ✓ | ✓ | 8 / 9 |
| DTD | ✓ | ✓ | x | x | ✓ | ✓ | x | ✓ | ✓ | 6 / 9 |
| CUE | ✓ | ✓ | ✓ | ✓ | x | ✓ | ✓ | ✓ | ✓ | 8 / 9 |
| Avro | ✓ | ✓ | x | x | ✓ | ✓ | x | x | ✓ | 5 / 9 |
| JTD | ✓ | ✓ | x | x | x | ✓ | x | ✓ | ✓ | 5 / 9 |
| TypeSchema | ✓ | ✓ | ✓ | x | ✓ | ✓ | ✓ | ✓ | ✓ | 8 / 9 |
| protobuf | ✓ | ✓ | x | x | x | ✓ | x | x | ✓ | 4 / 9 |
| GraphQL schema | ✓ | ✓ | ✓ | x | ✓ | ✓ | x | x | ✓ | 6 / 9 |

This is a helpful feature for our approach as often not all possible values are valid for specific fields in configuration files.

h) *Conditions* — This is fulfilled if the schema language provides the possibility to define conditional dependencies between fields. This is a advanced feature that is helpful because it allows to express for example that a particular field must be given only if another field has a specific value.

i) *References* — This is fulfilled if the schema language provides the possibility to define reusable subschemas that can be referenced in other parts of the schema. This is often useful in practice to reuse common data structures.

### B. Evaluation results

Tables I and II show the results of our evaluation. We come to the conclusion that JSON schema is sufficiently popular and expressive that we choose to use it as the schema language for our approach. The other schema languages are either less expressive or less popular. This result is in line with the work of Baazizi et al. [4], who also found over 80.000 JSON schema files on GitHub, and with their claim that JSON schema is the de-facto standard for JSON schema languages.

## IV. DESIGN

### A. User Interface

The design of *MetaConfigurator* is strongly inspired by another tool by one of the authors, which is a configurator GUI for the Minecraft server plugin BossShopPro [40], where the GUI components are generated based on a rudimentary custom schema language and several inbuilt schemas. That tool provides the user a text editor panel for editing configuration files of that domain in a text editor, as well as a GUI editor panel, where the user can edit their configuration file using GUI components. [41] *MetaConfigurator* differs from that tool by being generic, instead of being bound to a certain domain, by having a much more expressive schema language, a schema editor and many other features that improve the user experience, such as a search functionality.

Before we dive into the architecture and detailed design of the tool, this section provides an overview of the tool from the view of the user.

The user interface has three distinct views:

1) File Editor (figure 7): In this view the user can modify their configuration file, based on a schema.
2) Schema Editor (figure 8): In this view the user can modify their schema.
3) Settings (figure 15 in appendix): In this view the user can adjust parameters of the tool.

Figures 13 and 14 (appendix) provide a sketch of the planned user interface design before the implementation.

The UI is divided into two main panels: the *Code panel* (on the left) and the *GUI panel* (on the right). In the *Code panel* the user can modify their data by hand, as in a regular code 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 schema which the user provides (more on that in the next paragraphs). For example, for enum properties, the user will get a dropdown menu with the different options to choose from and for boolean properties the user will get a checkbox. Other features, such as tooltips that display the description and constraints of a

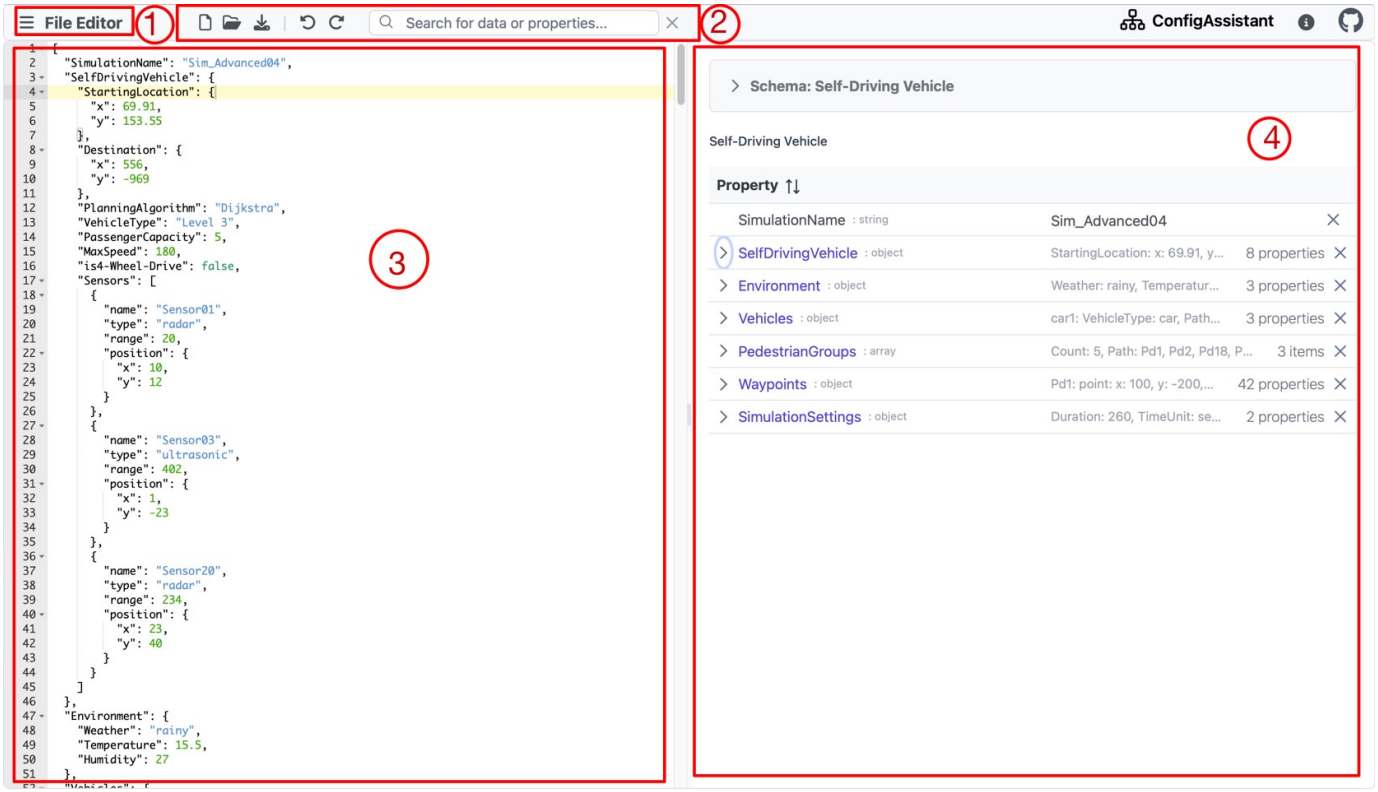


Fig. 7: UI of File Editor view. Different components highlighted in red: 1) button to switch to other view (e.g. to Schema Editor view), 2) Toolbar with various functionality, 3) Code panel, 4) GUI panel

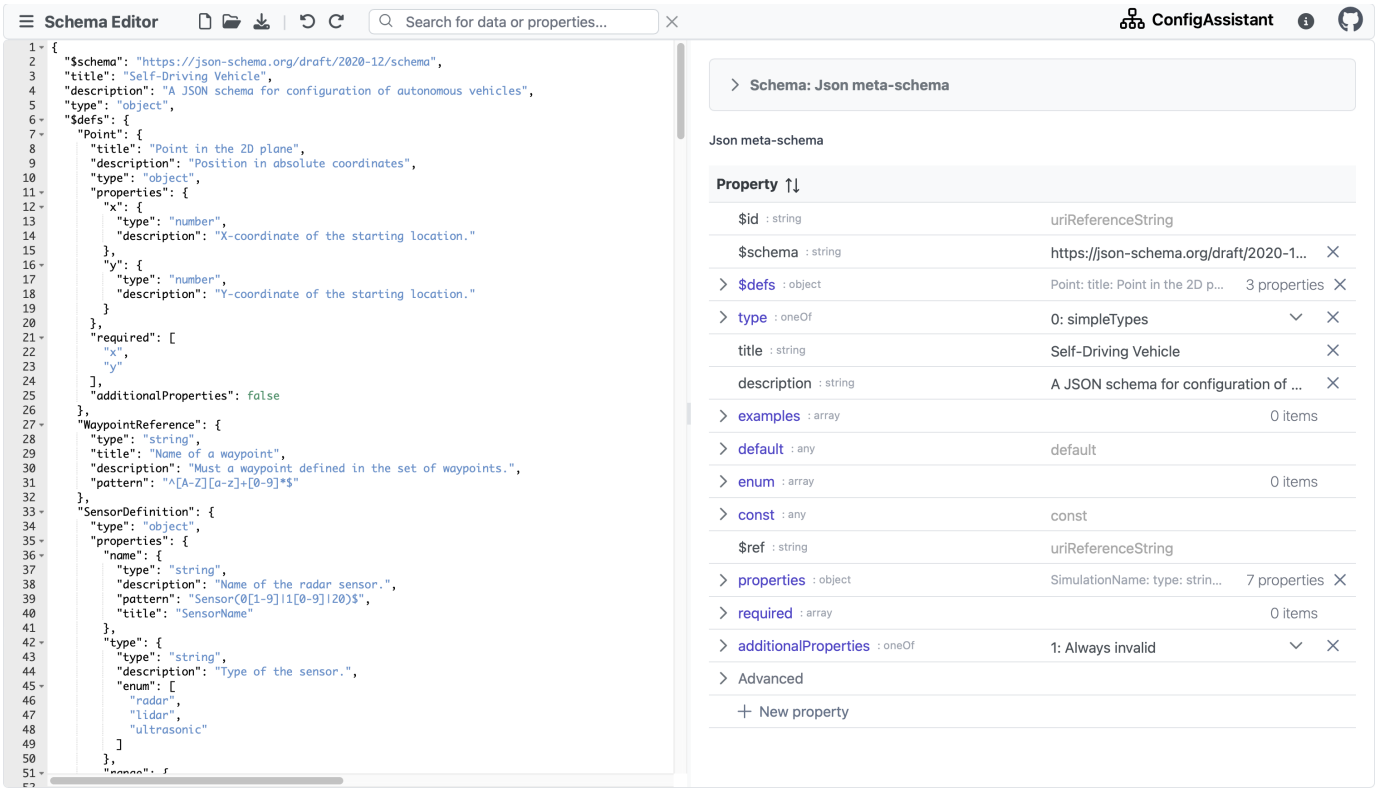


Fig. 8: UI of Schema Editor view

TABLE III
FILE DATA AND SCHEMA FOR THE DIFFERENT VIEWS

| View | Effective File Data | Effective Schema |
|---|---|---|
| File editor | User data | User schema |
| Schema editor | User schema | JSON Meta Schema |
| Settings | Settings data | Settings schema |

property, further assist the user.

This design combines the benefits of both a rich text code editor (efficient for many tasks, more suited for users with technical understanding of the data structure) with the benefits of a GUI (enables users without deep technical understanding to work with the data, assists also expert users).

As a schema is a configuration file itself, it can be treated as such and the tool can offer assistance accordingly. Note that whenever the user edits a configuration file using the tool, they do so using some underlying schema. Even the tool settings can be seen as a configuration file, for which the underlying schema is a settings schema.

Table III illustrates how for the different views, file data and schema being used by the tool differ.

### B. Intended Workflow

1) Upon initial access of *MetaConfigurator*, a dialog is displayed, where users can select their desired schema.
2) After selecting a schema, the user will find that a GUI is automatically generated on the right-hand side of the File Editor, tailored to the selected schema.
3) Through the GUI panel, users are assisted in creating or modifying configuration files.
4) If a user wishes to modify the selected schema, such as adding new properties, they can do so through the schema editor. Changes can be made using either the GUI panel or the code panel, and these modifications will automatically reflect in the file editor.

### C. Architecture

This section describes our main architectural design decisions. Those will not be relevant or visible for the user of the tool, but are important for understanding our implementation. Aim of those design decisions is to ensure modularity and maintainability of the tool.

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 that the user of the tool can access to modify the config data in an indirect way. It might be implemented as a code editor, a graphical user interface or any other way in which the data can be presented to the user. All editor panels are independent and do only have access to the data store but not to each other. Every editor panel subscribes to the changes of the data store, so it can be updated accordingly whenever the data in the store is changed. Additionally, every panel has the capabilities of updating the data store themselves, which is done when the user modifies the data in the editor panel. The following artificial example use-cases illustrate the capabilities of this architecture:

- Format converter: one panel shows the data in a code editor in JSON format, a second panel shows the data in a code editor in YAML format. Any semantic data change on one panel will cause the same semantic change in the other panel.
- Split-Screen Editor: one panel shows the data in a code editor, a second panel shows the data in a GUI. This way the user can have the efficiency of a text editor, but also the assistance of a GUI at the same time. Any semantic data change on one panel will be forwarded to the other panel.
- The Split-Screen Editor could be implemented for different data formats, such as YAML, JSON and XML. The architecture allows any data format as long as there exists a mapping from this data format to a JavaScript Object and back.

In practice, we implement only one code editor panel, as well as one GUI editor panel. The architecture, however, would be flexible enough to allow replacing any of these panels or adding new ones, since they are decoupled from each other and only communicate with the single source of truth data store.

*1) Single Source of Truth Data Store:* This is the core of the tool. The panels can subscribe to this store to receive updates whenever data is changed. Also, panels can trigger changes of the data in the store. Besides the current configuration data, the store also stores the path of the currently selected data entry and the schema that is currently being used.

*2) Code Panel:* For the code panel, we embed a rich-text code editor that already supports syntax highlighting and other useful features. We enable validation of whether the text is well-formed according to the JSON/YAML/XML Standard and add schema validation. 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 code editor with the new serialized data. The action of replacing the text in the code editor will cause formatting and comments to be lost, which we accept. In the future, several mechanism could be applied to avoid the loss of formatting or comments (see section VII-B).

When the user edits the text in the code 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.

To enable communication with the store, for any data format that the tool should support, we need a function to stringify a JavaScript object to a string in the data format and a function to parse a string in that data format as a JavaScript object.

To make it possible to highlight certain lines in the editor as erroneous (schema violations) or jump to certain lines (e.g. when the user selects a property in the GUI editor, we want to jump to the same property in the text editor), we need a function `determineRow(editorContent, dataPath)` that can determine the corresponding editor line, based on the configuration text and a given data path.

The other way around, when the user places their cursor inside the text editor, we want to determine the path of the element that the cursor is currently at. This requires a function `determinePath(editorContent, cursorPosition)` which returns a data path based on the configuration text and a given cursor position.

*3) GUI Assistance Panel:* The GUI assistance panel directly works with the given schema and provides the user with corresponding GUI elements, such as a checkbox for a boolean data structure or a text field for a string data structure. Additional GUI elements, such as tooltips (showing the description of a data field) are used to support the easier. The GUI elements are constructed in the following manner: a schema is seen as a hierarchical tree of data field definitions and their corresponding constraints. A data field can either be simple (string, boolean, number, integer) or complex (array or object with children). Every schema has a root data field. The GUI element for this root data field is constructed. When constructing the GUI element for a complex data field, all GUI elements of the child data fields are constructed too, in a recursive manner. This way, the whole schema tree is traversed and GUI elements for all entries are constructed. To avoid overwhelming the user with too many GUI elements, the ones with child elements can be expanded or collapsed by the user and only a limited amount of them is expanded by default. By design, each of these constructed GUI elements is mapped to their corresponding data field (in other words: to a path in the data structure). The initial values of all GUI elements are taken from the data in the store, by accessing the data at the given paths. Whenever the values in a GUI element are adjusted by the user, the data in the store will be updated with the new values.

## V. Implementation

This section contains the implementation details of *MetaConfigurator*. We first describe the implementation and features of the two main components of *MetaConfigurator*, the code editor and the GUI editor. Then, we explain the schema preprocessing steps that are required to generate the GUI editor. Finally, we describe how we developed a new meta schema for JSON schema that is more suitable for our tool.

### A. Technologies

We use vue.js [42] as the UI framework for our tool, combined with the component library PrimeVue [43]. A detailed list of all libraries used can be found in the wiki of our GitHub repository[1].

### B. Code Panel

The code editor is a GUI panel designed for editing the configuration files. For this project, we use the *Ace Editor* [44] library to embed an interactive code editor into our user interface. It provides useful features for our approach, such as syntax highlighting and code folding. To make our code editor more user-friendly, we implemented several features, which are described in the following.

[1]https://github.com/PaulBredl/config-assistant/wiki/All-libraries

*1) Schema Validation:* 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* [45] library, which supports the newest JSON schema draft 2020–12. If schema violations are found, the corresponding user data lines in the code editor will be marked with a red error hint, which also describes the violation.

*2) Linkage of text with the data model:* As described in section IV-C2, to map a cursor position in the text editor to a path in the data model we need to implement the function `determinePath(editorContent, cursorPosition)` and to map a path in the data model to a text row in the editor, we need to implement the function `determineRow(editorContent, dataPath)`.

For *JSON*, the functions have been implemented using a *Concrete Syntax Tree (CST)*. The text content is parsed as a CST@. Then this tree is traversed recursively. Every tree node has a range property, describing the start and end index of the text belonging to the node. To determine the corresponding path for a cursor position, the cursor position is translated to a character index `targetCharacter` within the text. Then the CST is traversed and for all nodes `currentNode` of type array or object for which `targetCharacter` $\in$ `currentNode.range`, the child nodes are checked, and the key of the node (or index for array elements) is appended to the result path. This way, the corresponding path is built up.

To determine the cursor position for a given path, we reverse this algorithm: the CST is traversed until the node is found whose path matches the target path. Then `currentNode.range.start` is returned as the result index, which is then translated into a cursor position (row and column).

For *YAML*, this linkage is not yet implemented and will be part of further work.

*3) Editor Operations:* The code 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.

### C. GUI Panel

The GUI editor is a component that allows the user to edit the configuration data in a GUI, which is generated based on the schema of the configuration data. 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 7 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 V-D. To assist the user in editing the configuration data, the GUI editor offers a set of features, which are described in the following.

*1) 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

TABLE IV
CORRESPONDING GUI IMPLEMENTATIONS FOR JSON SCHEMA FEATURES

| JSON schema type / keywords | GUI implementation |
|---|---|
| string | Text field. |
| number | Text field which allows only floating point numbers and has buttons to increment and decrement. |
| integer | Text field which allows only integer numbers and has buttons to increment and decrement. |
| boolean | Checkbox. |
| object | Expandable list of child columns in the properties table (see figure 9). |
| array | Similar as for object. Also has a button to add new items (see figure 10). |
| enum | Dropdown menu. |
| const | Dropdown menu with just one entry. |
| required | Red asterisk left 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. |

or array. This will show the sub-properties of the object or the elements of the array. We limit the depth of the data tree to a configurable value, to prevent the GUI editor becoming too overwhelming. However, 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 which path the GUI editor currently shows and to navigate back to upper levels of the tree.

*2) GUI implementations for JSON schema features:* Table IV shows how the different JSON schema features are implemented in our GUI. In our GitHub wiki[2], we provide a detailed overview of which JSON schema features are supported by our tool.


Fig. 9: GUI implementation for object properties


Fig. 10: GUI implementation for array properties

*3) 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 shown if the property is not required and there exists data.

[2]https://github.com/PaulBredl/config-assistant/wiki/JSON-schema-keyword-support

*4) Schema Information Tooltip:* When the user hovers over the property key or array index, an overlay is displayed (figure 11), which contains all information from the schema about that property. We manually implemented a generation of a textual description for each of the JSON schema keywords. Starting with the title and description of the property, the overlay then shows constraints (such as the number must be greater than 0) and at the bottom, it also shows schema violations, in case there are any. This feature helps the user to understand the constraints and the meaning of a property.

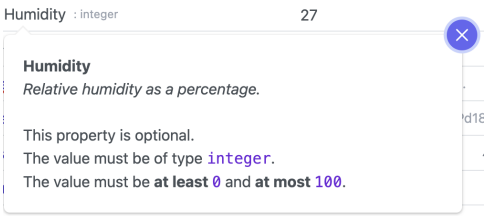


Fig. 11: Tool-tip

*5) Highlighting Schema Validation Errors:* When the configuration data does not comply with the schema, the corresponding 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. Additionally, the schema information tooltip lists all schema violations, as shown in figure 12.

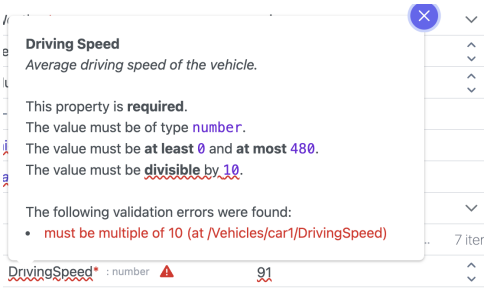


Fig. 12: Tool-tip with schema violation

### D. 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 represents a type union. Here there is no obvious GUI component that can represent this. 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 that is loaded into the schema editor. The following sections describe the preprocessing steps in detail and how we solve cases like the one just mentioned.

*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 `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 `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.

*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. In the following, we describe the preprocessing steps in detail.

*a) Resolving references:* JSON schema uses the `$ref` keyword to reference other schemas. This can either be references to schemas in the same file (using the `$defs` keyword), references to other local files, or references to schemas at a URL in the web. We currently only support references to schemas in the same file. These are lazily resolved as the first preprocessing step. Listing 3 shows an example schema, Listing 4 shows the equivalent example after this first preprocessing step.

*b) Resolving allOfs:* The `allOf` keyword in JSON schema specifies that all the schemas in the given array must be valid. To simplify any other operation on the schema, we aim to merge the schemas in the `allOf` array into one equivalent schema. As the first step, we do a recursive step by preprocessing all the schemas of the `allOf` array. Then, we use the *mergeAllOfs* [46] library to merge all the sub-schemas. Listing 5 shows the previous example schema after this step. It is important to note that this library only supports a few keywords of JSON schema, most notably the `properties` and `items` keyword. Hence, the *MetaConfigurator* has only limited support for `allOf` and any other keywords for which we use this library in the preprocessing.

*c) Converting types to oneOf:* In JSON schema, a property can have multiple types, such as shown in listing 6. A semantically equivalent schema can be generated by the use of `oneOf`, where each sub-schema contains exactly one of

```
{
  "title": "NonEmptyString",
  "$ref": "#/$defs/nonEmptyString",
  "$defs": {
    "nonEmptyString": {
      "type": "string",
      "minLength": 1
    }
  }
}
```

Listing 3: Simple JSON schema before reference resolving

```
{
  "allOf": [
   {
     "title": "NonEmptyString"
   },
   {
     "type": "string",
     "minLength": 1
   }
  ],
  "$defs": {
    "nonEmptyString": {
      "type": "string",
      "minLength": 1
    }
  }
}
```

Listing 4: Simple JSON schema after reference resolving

```
{
  "title": "NonEmptyString",
  "type": "string",
  "minLength": 1,
  "$defs": {
    "nonEmptyString": {
      "type": "string",
      "minLength": 1
    }
  }
}
```

Listing 5: Simple JSON schema after allOf resolving

the types, as shown in listing 7. If a schema defines more than one type, we convert the types to `oneOf`. As `oneOf`s are represented as a dropdown menu in the GUI editor, we now have a way to represent multiple types in the GUI. For schemas that already contain `oneOf`, every type is *multiplied* by every existing `oneOf` sub-schema. For two types and three `oneOf` sub-schemas, this will result in a new `oneOf` with six options. An exception is when a type can not be merged with a `oneOf` sub-schema (e.g., the `type` is "boolean" and the `oneOf` sub-schema has type "string"). In that case, the incompatible pair is dismissed.

```
{
  "type": ["object", "boolean", "string"]
}
```

Listing 6: Simple JSON schema with three possible types

```
{
  "oneOf": [
    {
      "type": "object"
    },
    {
      "type": "boolean"
    },
    {
      "type": "string"
    }
  ]
}
```

Listing 7: Simple JSON schema after conversion of types to oneOf

*d) Removing incompatible oneOfs and anyOfs:* A schema may have `oneOf` or `anyOf` options that are not compatible with the schema of the property (e.g., sub-schemas that can never be fulfilled in combination with the property schema). Listing 8 provides an example of a schema with an incompatible `oneOf` option. For every `oneOf` and `anyOf` sub-schema, we check whether it can be merged with the schema of the property. The options which are not compatible are removed (see listing 9).

```
{
  "type": "object",
  "oneOf": [
    {
      "type": "object"
    },
    {
      "type": "boolean"
    }
  ]
}
```

Listing 8: Simple JSON schema with incompatible oneOf option

*e) Merging singular oneOfs and anyOfs:* Because of the previous pre-processing step, it can happen that for some `oneOf`s or `anyOf`s, there remains only one compatible sub-schema left (see listing 9). If this is the case, the use `oneOf`/`anyOf` is redundant, as that singular sub-schema must be chosen implicitly. Therefore, if there exists only one singular choice for `oneOf`/`anyOf`, we merge its sub-schema into the property schema and remove the use of `oneOf`/anyOf (see listing 10).

```
{
  "type": "object",
  "oneOf": [
    {
      "type": "object"
    }
  ]
}
```

Listing 9: Simple JSON schema with incompatible oneOf option removed

```
{
  "type": "object"
}
```

Listing 10: Simple JSON schema with singular oneOf merged into property schema

*f) Attempting to merge oneOfs into anyOfs:* Schemas can use both `anyOf` and `oneOf` at the same time. Especially after converting type unions to `oneOf`, it happens that a schema has `oneOf` options (typically for types) and simultaneously anyOf options. The user will then have to select both a `oneOf` sub-schema, as well as an `anyOf` sub-schema in the GUI. We observed a special scenario in the JSON meta schema, where the `oneOf` selection was always implicitly given by the `anyOf` selection. For every single `anyOf` sub-schema, only one `oneOf` sub-schema was compatible. In that scenario, we can merge the `oneOf`s into the `anyOf`s: for every `anyOf` sub-schema, we merge the one compatible `oneOf` sub-schema into it. This is precisely what this pre-processing step does: if possible, the `oneOf` sub-schemas are merged into the `anyOf` sub-schemas, and the `oneOf` property is removed from the schema.

*g) Preprocessing oneOfs and anyOfs:* For all remaining `oneOf` and `anyOf` sub-schemas, the internal pre-processing steps are executed.

*3) Calculating an 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, we evaluate the data and dependent on validity or not, we either use the `then` or the `else` schema. We similarly handle 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.

```
{
  "properties": {
    "mode": {
        "enum": ["manual", "automatic"]
    }
  },
  "if": {
    "properties": {
      "mode": {
        "const": "manual"
      }
    },
    "required": ["mode"]
  },
  "then": {
    "properties": {
        "manualValue": {
            "type": "number"
        }
    }
  }
}
```

Listing 11: Data dependent schema. If the field `mode` is set to "manual" in the data, users will expect that the GUI shows the `manualValue` property

```
{
  "properties": {
    "mode": {
        "enum": ["manual", "automatic"]
    },
    "manualValue": {
        "type": "number"
    }
  }
}
```

Listing 12: Effective schema when the value for `mode` is "manual"

Listing 11 shows an example of a data-dependent schema and listing 12 shows the effective schema when the value for `mode` is "manual".

### *E. Developing a Custom Meta Schema*

The schema editor page has the same structure as the File editor page, as discussed in previous sections. The only difference is that the schema used for generating the GUI panel is not the schema file provided by the user but the Json schema meta schema, i.e., the schema that defines the structure of valid JSON schema files. However, applying our generic approach on the official JSON schema meta schema [3] does not result in a user-friendly editor for creating and modifying schema files. In this section we discuss the reasons for that and how we developed a new meta schema that circumvents the problems of the official meta schema.

*1) Missing descriptions:* With the `description` keyword, schema authors can give descriptions to any elements of their schema. This can help the user of a schema in many ways, for example, the author can specify the unit of a numeric field or give other additional information. The JSON schema meta schema does not provide any descriptions. Users, especially those without prior knowledge in JSON schema, might not understand the meaning of the fields of JSON schema. Thus, we insert descriptions from the JSON schema specification [3] into our modified meta-schema.

*2) External references: MetaConfigurator* does not support references to external schemas yet, i.e., references inside the schema to a schema at a specific URL. Also, *MetaConfigurator* does not support the `$vocabulary` keyword. Both features are used in the JSON schema meta schema, as it is distributed over multiple schema files. To circumvent that problem, we put all schemas in one schema file into the `$defs` object and replace the external references with local references.

*3) Use of dynamic anchors and references:* The JSON schema meta schema uses dynamic references and dynamic anchors. The difference of those keywords in comparison to the `$ref` keyword is that they provide a way to dynamically extend the JSON meta schema at runtime. For example, one could combine the JSON schema meta schema with an extension that defines how fields should be serialized in XML. We do not support dynamic references and anchors yet. We replaced all of them by "non-dynamic" references using the `$ref` keyword.

*4) Allowing each field in each context:* The JSON schema meta schema allows each field in each context. For example, if the `type` keyword is used and set to `string`, then the `properties` keyword is allowed, even though it does not make sense in that context. According to the specification, any validator should ignore the fields that do not make sense in the current context. Consequently, the user does not need to see those fields, but instead, the user gets overwhelmed by the amount of fields and does not know which fields are relevant for the current context. This is also a feedback we got from our user study.

```
{
  "if": {
    "$ref": "#/$defs/hasTypeArray"
  },
  "then": {
    "$ref": "#/$defs/arrayProperty"
  }
}
```

Listing 13: If condition for array properties. The `hasTypeArray` is valid if the current property is of type array. The `arrayProperty` schema defines the properties of an array.

Thus, we added `if` conditions to each field to only show them when they make sense in the current context. Listing 13 shows an example of such an `if` condition. The relevant properties for arrays are only shown when the current property is of type array.

To even more reduce the amount of fields shown to the user, we also introduced a custom keyword just for our own meta schema. The keyword `advanced` is a boolean field that is set to `false` by default. It is wrapped in an `metaConfigurator` object, which is ignored by any validator as it is not part of the JSON schema specification. We use this wrapper to prevent any other schema extensions from colliding with our keyword. When set to `true`, the field

not shown by default, but only when the user expands the advanced section. We put all fields that are not required for the basic usage of the schema into the advanced section. For this, we oriented ourselves on the work of Baazizi et al. [4], which analyzed the usage of JSON schema keywords in 82,000 JSON schemas.

## VI. User Study

We conduct a qualitative user study with five participants.

During each interview, we introduce *MetaConfigurator*, give the participant tasks to execute using the tool and finish the session with open ended questions. We observe how the participants work with the tool and which difficulties they have when executing the tasks. Additionally, we ask them for feedback and improvement suggestions. We also conduct a survey to gather demographic information about the participants of the user study.

Note that besides the user study we have applied *MetaConfigurator* on several schemas (such as EnzymeML [**?**] and the Strenda schema [**?**]) and configuration files from the real world, to verify that it works and does bring benefits to the user.

### A. Research Questions

We intend to address the following research questions with the user study:

1) **RQ1:** Which aspects of the tool can be improved?
2) **RQ2:** Are users able to perform the followings types of tasks using the tool:
   - **RQ2.1** Retrieve information from configuration files in the context of a given schema
   - **RQ2.2** Modify configuration files within the constraints of a given schema
   - **RQ2.3** Modify a schema file
3) **RQ3:** Would people use the tool in practice?

### B. Methodology

*a) Potential Users:* We look for potential users to conduct the interviews. There are two basic standards we follow to find a potential user:

- Professors and students who are interested in our application.
- People who frequently use configuration filesand schema languages

*b) Interview Questions:* For the interview, we created a JSON schema and configuration file, about a made-up self-driving car simulation, The interview questions deal with working with those files. We divide our interview tasks into four parts:

- Setup: Open the application and open schema and configuration file.
- Information Retrieval Questions: with increasing difficulty, the participant has to retrieve information from the configuration file. Much easier to solve by using the GUI panel.
- Configuration Modifications: Different tasks that involve changing the configuration file. Intend is that the participant becomes more familiar with the GUI panel.
- Schema modifications: Making adjustments to the schema. Most difficult, but still feasible using the GUI panel.

The interview tasks can be found in the appendix in section B-A.

*c) Interview Process:* We proceed our interview with one interviewee each time. The interview lasts around one hour. At the beginning, the interviewee is asked about approval of recording, and they can stop participating at any time. We introduce *MetaConfigurator* to the interviewee. Then we send the participant the tasks, an example configuration file and let them work on the tasks while sharing their screen. During this task solving session, we provide the interviewee with some basic help if they ask specific questions about the tool. The interview is recorded, and we make notes of the answers, feedback and behavior of the interviewee. Finally, in an open dialog, we ask the interviewee about more feedback and improvement suggestions as well as their opinion on the tool.

*d) Survey:* After the interview, we ask each interviewee to fill out a survey (follow-up questions). This survey covers the following points:

- Occupation of the interviewee.
- Experiences in software engineering,
- Frequency of working with JSON/YAML files.
- Domains in which people use with JSON/YAML files.
- Feedback about the application after interview.

### C. Results

*1) **RQ1:** Which aspects of the tool can be improved?:* Tables VI-X (in the appendix) show the feedback of the interviewees, as well as which measures we took based on it.

*2) **RQ2:** Are users able to perform the followings types of tasks using the tool?:* Table V (in the appendix) shows the accuracy and difficulties that the participants had when solving the tasks. Accuracy is determined as the ratio of tasks that were solved correctly without the need of any hints to the total number of tasks. With the help of hints, all tasks could be completed by every participant.

*3) **RQ3:** Would people use the tool in practice?:*

### D. Threads to Validity

Our user study is qualitative and only has a small sample size of 5 participants. Hence, the results are not representative and cannot be generalized. In addition, the tasks of the user study do only cover a small subset of operations that a user may want to perform with *MetaConfigurator* to edit configuration files and schemas. Due to the time constraints of the user study, we only cover rather simple tasks. Thus, for very complex tasks, the user study is not conclusive. For the purpose of getting feedback on the tool and finding out whether it is useful in practice, these limitations are acceptable. For RQ2, we cannot generalize our results but only provide a first impression of how users work with *MetaConfigurator*.

## VII. Discussion

This section discusses the implications of our work and future work.

### A. Implications of our work

*MetaConfigurator* provides a novel approach for editing configuration files by combining the advantages of a GUI and a code editor Our user study suggests that *MetaConfigurator* is easy to use and intuitive and that it has practical use cases in many domains. However, the user study also revealed some limitations of *MetaConfigurator*. Editing schemas with *MetaConfigurator* is not as intuitive as editing configuration 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.

### B. Future work

To make *MetaConfigurator* more useful for users who are not familiar with JSON schema, there are several possible improvements. First, a visual schema editor could be added to *MetaConfigurator*, similar to schema editors discussed in section II-B2. These would provide a graph view of the schema, which is easier to understand than the JSON schema in text form or the tree view in *MetaConfigurator*. Second, *MetaConfigurator* could provide more guidance for users who are not familiar with JSON schema, e.g., by providing an interactive tutorial or supporting a less complex schema language.

There are many other possible improvements to *MetaConfigurator*. A desktop version of *MetaConfigurator* could be developed, which would allow users to edit files on their local machine, which is more convenient than loading them into the web application. Similarly, integration in other tools, such as IDEs, could be helpful for many users. *MetaConfigurator* currently only supports JSON schema draft 2020–12, but it could be extended to support other drafts by converting imported schemas to the latest draft. Furthermore, YAML is not fully supported yet, which could be added in the future. Another point that can be addressed is the loss of formatting and comments in YAML documents, when they are updated with new data. This could be avoided by replacing only the section in the YAML document that corresponds to the change, instead of replacing the complete document. To allow for different styles of formatting, the user could be provided with global formatting style settings (such as level of indentation or whether in YAML strings should be in quotation marks or not). To deal with the loss of comments, a technique that keeps track of any comments in the text and then restores them after the text is replaced could be implemented. This has already been done in another tool of one of the authors [41].

Finally, *MetaConfigurator* could be extended to support code generation, e.g., for generating Java classes from a JSON schema, which is useful for developers.

To improve the user study, it could be repeated with more participants, so that the results are more representative. Instead of just having participants solve tasks, it could also be interesting to have one group of participants solve tasks with *MetaConfigurator* and another group solve the same tasks with only a text editor. This way, we could evaluate whether *MetaConfigurator* is actually more efficient than just using a text editor.

## VIII. Conclusion

This paper addresses the development of *MetaConfigurator*, a tool that generates YAML/JSON file editor GUIs tailored to a given data schema. According to the criteria expressiveness and popularity, we choose JSON schema as schema format for the tool. The tool is successfully implemented and our user study shows that it can be applied to solve practical tasks in 1) retrieving information from configuration files in the context of a schema, in 2) modifying configuration files and in 3) editing schemas. The interest and positive feedback of the participants suggests that *MetaConfigurator* will be applied in practice.

## Acknowledgments

This should be a simple paragraph before the References to thank those individuals and institutions who have supported your work on this article.

## REFERENCES


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

[2] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of json schema," in *Proceedings of the 25th International Conference on World Wide Web*, ser. WWW '16. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2016, p. 263–273. [Online]. Available: https://doi.org/10.1145/2872427.2883029

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

[4] M.-A. Baazizi, D. Colazzo, G. Ghelli, C. Sartiani, and S. Scherzinger, "An empirical study on the "usage of not" in real-world json schema documents (long version)," 2021.

[5] M. Kristensen, "JSON Schema Store — schemastore.org," https://www.schemastore.org/json/, [Accessed 07-10-2023].

[6] G. Barbaglia, S. Murzilli, and S. Cudini, "Definition of rest web services with json schema," *Software: Practice and Experience*, vol. 47, no. 6, pp. 907–920, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2466

[7] I. C. Siffa, J. Schäfer, and M. M. Becker, "Adamant: a JSON schema-based metadata editor for research data management workflows," *F1000Research*, vol. 11, p. 475, Apr. 2022. [Online]. Available: https://doi.org/10.12688/f1000research.110875.1

[8] J. Bosak, T. Bray, D. Connolly, E. Maler, G. Nicol, M. Sperberg-McQueen, L. Wood, and J. Clark, "W3c xml specification dtd ("xmlspec")," 1998. [Online]. Available: https://www.w3.org/XML/1998/06/xmlspec-report-19980910.htm

[9] D. Fallside and P. Walmsley, "Xml schema part 0: Primer second edition - w3c recommendation 28 october 2004," 2004. [Online]. Available: https://www.w3.org/TR/2004/REC-xmlschema-0-20041028/

[10] G. J. Bex, F. Neven, and J. Van den Bussche, "Dtds versus xml schema: A practical study," in *Proceedings of the 7th International Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS 2004*, ser. WebDB '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 79–84. [Online]. Available: https://doi.org/10.1145/1017074.1017095

[11] D. Lee and W. W. Chu, "Comparative analysis of six xml schema languages," *SIGMOD Rec.*, vol. 29, no. 3, p. 76–87, sep 2000. [Online]. Available: https://doi.org/10.1145/362084.362140

[12] W. Martens, F. Neven, M. Niewerth, and T. Schwentick, "Bonxai: Combining the simplicity of dtd with the expressiveness of xml schema," *ACM Trans. Database Syst.*, vol. 42, no. 3, aug 2017. [Online]. Available: https://doi.org/10.1145/3105960

[13] "Configure unify execute," accessed 18-May-2023. [Online]. Available: https://cuelang.org/

[14] "What is apache avro?" accessed 14-June-2023. [Online]. Available: https://www.ibm.com/topics/avro

[15] U. Carion, "JSON Type Definition," RFC 8927, Nov. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8927

[16] C. Kappestein, "Typeschema," 2023. [Online]. Available: https://typeschema.org/

[17] O. Hartig and J. Hidders, "Defining schemas for property graphs by using the graphql schema definition language," in *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, ser. GRADES-NDA'19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3327964.3328495

[18] "Protocol Buffers — protobuf.dev," https://protobuf.dev, [Accessed 01-May-2023].

[19] B. H. A.Wright, H.Andrews Ed, "validation," 18 December 2022, accessed 18-May-2023. [Online]. Available: https://json-schema.org/draft/2020-12/json-schema-validation.html#name-validation-keywords-for-any

[20] B. H. A. Wright, H. Andrews, "Json schema validation: A vocabulary for structural validation of json," March 20, 2020, accessed 06-May-2023. [Online]. Available: https://json-schema.org/draft/2019-09/json-schema-validation.html

[21] baeldung, "Validate an xml file against an xsd file," Sep 2023. [Online]. Available: https://www.baeldung.com/java-validate-xml-xsd

[22] N. Perriault, A. Ramaswami, and N. Grosenbacher, "GitHub - rjsf-team/react-jsonschema-form: A React component for building Web forms from JSON Schema. — github.com," https://github.com/rjsf-team/react-jsonschema-form, 2023, [Accessed 08-10-2023].

[23] D. Jensen, M. J. Bennett, D. Dervisevic, C. Edwards, and M. Marcacci, "GitHub - json-schema-form/angular-schema-form: Generate forms from a JSON schema, with AngularJS! — github.com," https://github.com/json-schema-form/angular-schema-form, 2016, [Accessed 08-10-2023].

[24] D. Higgins and Icebob, "GitHub - vue-generators/vue-form-generator: :clipboard: A schema-based form generator component for Vue.js — github.com," https://github.com/vue-generators/vue-form-generator, 2019, [Accessed 08-10-2023].

[25] E. Müller, E. Neufeld, S. Dirix, L. Koehler, and F. Gareis, "More forms. Less code. - JSON Forms — jsonforms.io," https://jsonforms.io, 2021, [Accessed 08-10-2023].

[26] J. de Jong, "JSON Editor Online: JSON editor, JSON formatter, query JSON — jsoneditoronline.org," https://jsoneditoronline.org, [Accessed 08-10-2023].

[27] S. Zimmer, C. Chapellier, and F. Daoust, "GitHub - jsonform/jsonform: Build forms from JSON Schema. Easily template-able. Compatible with Bootstrap 3 out of the box. — github.com," https://github.com/jsonform/jsonform, 2021, [Accessed 08-10-2023].

[28] I. C. Siffa, J. Schäfer, and M. M. Becker, "Adamant: a json schema-based metadata editor for research data management workflows," *F1000Research*, vol. 11, 2022.

[29] "XML Editor: XMLSpy — altova.com," https://www.altova.com/xmlspy-xml-editor, [Accessed 08-10-2023].

[30] L. T. Limited, "JSON Schema Editor — liquid-technologies.com," https://www.liquid-technologies.com/json-schema-editor, [Accessed 08-10-2023].

[31] "XML editor and validator tool — xml-buddy.com," https://www.xml-buddy.com, [Accessed 08-10-2023].

[32] "JSON Schema editor for Windows — json-buddy.com," https://www.json-buddy.com/json-schema-editor.htm, [Accessed 08-10-2023].

[33] "XML Editor - XMLBlueprint — xmlblueprint.com," https://www.xmlblueprint.com, [Accessed 08-10-2023].

[34] "The complete solution for XML authoring, development and collaboration. — oxygenxml.com," https://www.oxygenxml.com/xml_developer.html, [Accessed 08-10-2023].

[35] F. Frasincar, A. Telea, and G.-J. Houben, *Adapting Graph Visualization Techniques for the Visualization of RDF Data*. London: Springer London, 2006, pp. 154–171. [Online]. Available: https://doi.org/10.1007/1-84628-290-X_9

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

[37] L. Deligiannidis, K. J. Kochut, and A. P. Sheth, "Rdf data exploration and visualization," in *Proceedings of the ACM First Workshop on CyberInfrastructure: Information Management in EScience*, ser. CIMS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 39–46. [Online]. Available: https://doi.org/10.1145/1317353.1317362

[38] C. North, N. Conklin, and V. Saini, "Visualization schemas for flexible information visualization," in *IEEE Symposium on Information Visualization, 2002. INFOVIS 2002.*, 2002, pp. 15–22.

[39] P. Carbonnelle, "TOP IDE Top Integrated Development Environment index — pypl.github.io," https://pypl.github.io/IDE.html, [Accessed 18-May-2023].

[40] F. Neubauer, "Bossshoppro - the most powerful chest gui shop/menu plugin." [Online]. Available: https://www.spigotmc.org/resources/bossshoppro-the-most-powerful-chest-gui-shop-menu-plugin.222/

[41] ——, "GitHub - logende/bossshopproeditor: powerful and user-friendly web application, which makes setting up shops for the bukkit plugin bossshoppro way easier. — github.com," https://github.com/Logende/BossShopProEditor, 2023, [Accessed 11-10-2023].

[42] E. You, "Vue.js - The Progressive JavaScript Framework — Vue.js — vuejs.org," https://vuejs.org, 2023, [Accessed 18-10-2023].

[43] T. Küçükoğlu, "PrimeVue — Vue UI Component Library — primevue.org," https://primevue.org, 2023, [Accessed 18-10-2023].

[44] F. J. Harutyun Amirjanyan, "Code editor," Mar 28, 2010, accessed 14-05-2023. [Online]. Available: https://ace.c9.io/

[45] E. P. etc., "A useful library for validating a json file based on the schema," May 17, 2015. [Online]. Available: https://ajv.js.org/

[46] M. Hansen, "GitHub - mokkabonna/json-schema-merge-allof: Simplify your schema by combining allOf — github.com," https://github.com/mokkabonna/json-schema-merge-allof, 2023, [Accessed 18-10-2023].

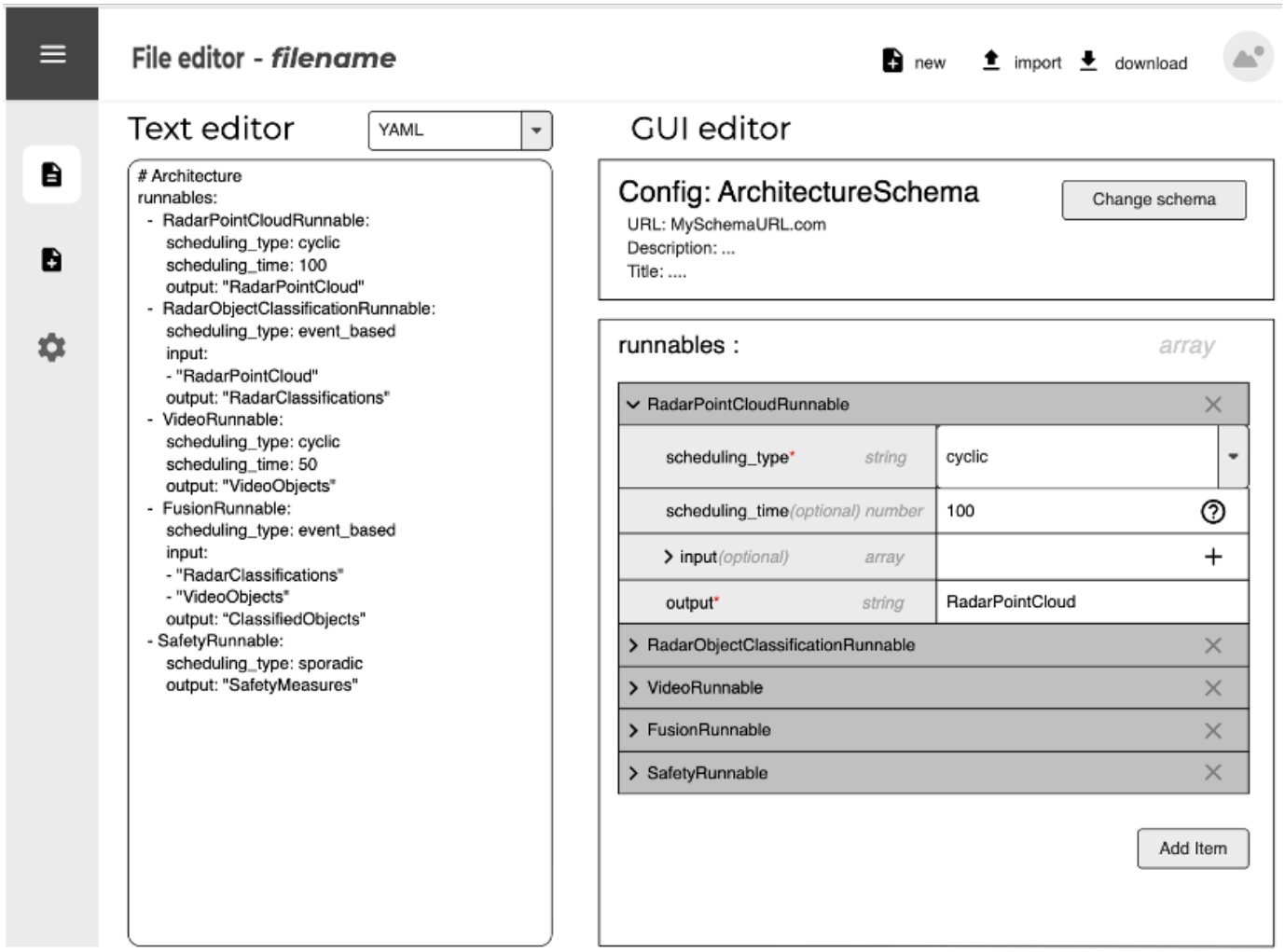


Fig. 13: Sketch of the Tool before the implementation. File Editor view.

## APPENDIX A
## DESIGN

## APPENDIX B
## USER STUDY

### A. Interview Tasks

This part is about the newest version of interview tasks we prepared to conduct our user study.
**Remark:** Task 3 was added after the first user study and also some details in other tasks were modified.

*1) Introduction:* For these tasks you are presented a schema that you have not seen or worked with before. We have prepared a schema of a made-up simulation software in which self-driving cars are simulated. There is one self-driving car that has to navigate from a start point to an end point but there are also other vehicles and pedestrians simulated. For these tasks, the exact details are not important.

*2) Task 1: Setup:*

1) Go to https://paulbredl.github.io/config-assistant/
2) Select the option “Select a Schema”: “Example schema”, then “Autonomous Vehicle Schema”.
3) Open the example configuration file we have sent to you. The following tasks will assume this schema and this example file.

*3) Task 2: Questions:*

1) What is the name of the simulation?
2) What is the weather in our simulated environment?
3) What is the total duration of the simulation?
4) Humidity is a subproperty of the Environment. Would 150 be a valid value for Humidity?

*4) Task 3: Modifying the configuration file:*

1) Change the name of the simulation to Sim_Advanced05.
2) The VehicleType of the self-driving car is currently “Level 3”. Change it to the highest possible level.
3) The configuration file has validation errors, i.e., it is not valid according to the schema.
   Find out what the errors are. Edit the file so it becomes a valid configuration.

*5) Task 4: Modifying the schema:*

1) For many applications it is good practice or even required that a schema has a unique identifier, which usually is a URL.

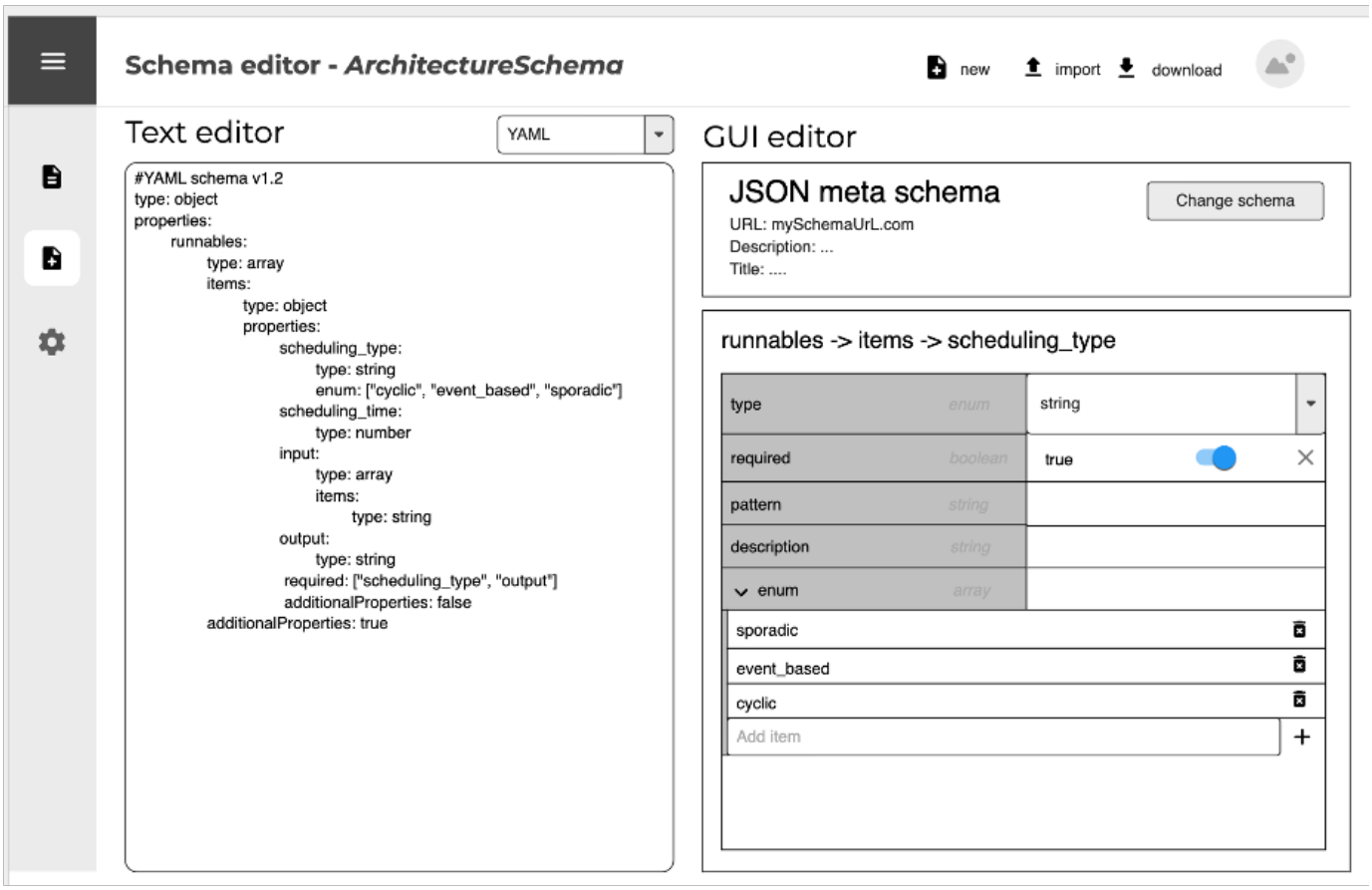


Fig. 14: Sketch of the Tool before the implementation. Schema Editor view.

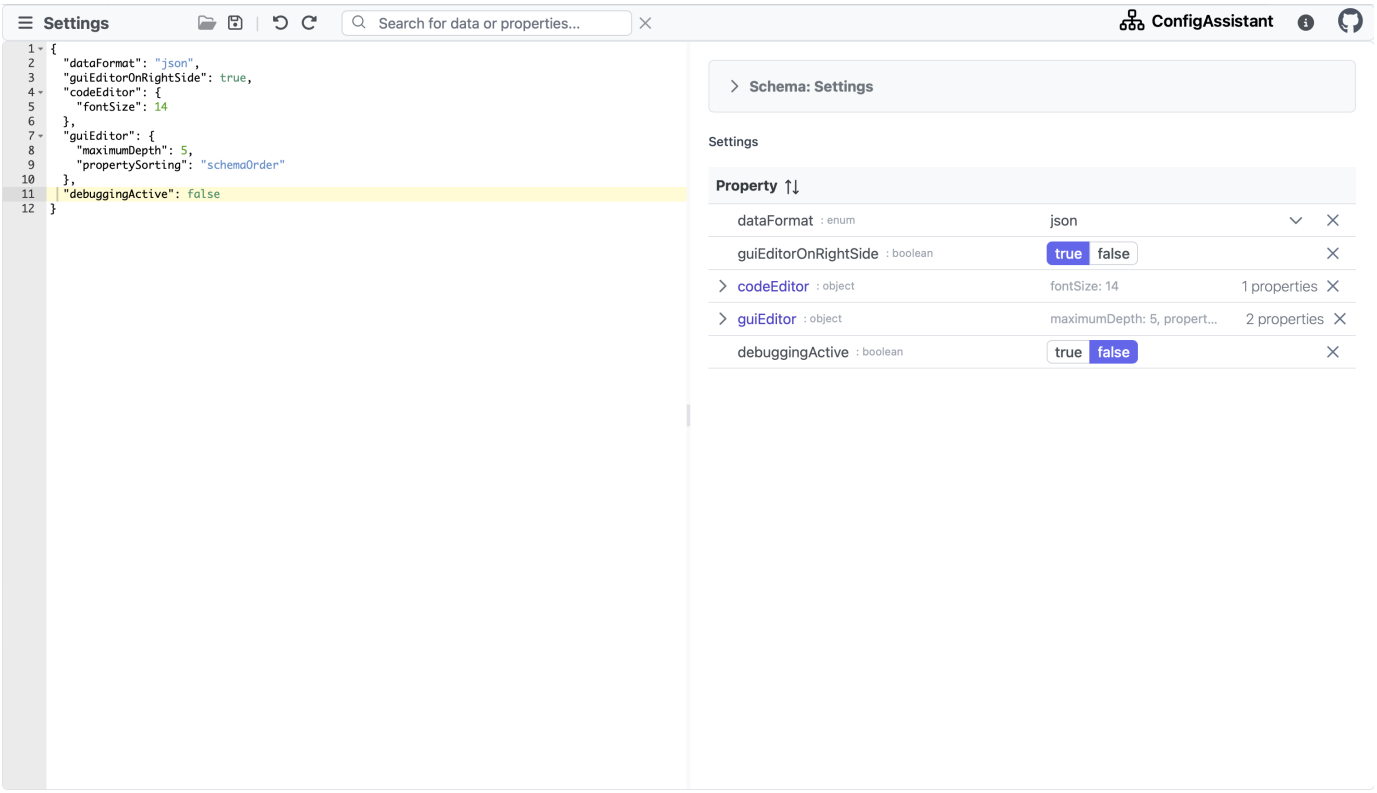


Fig. 15: UI of Settings view

TABLE V
RESULTS OF USER STUDY

| | Accuracy & Notes | Difficulties |
|---|---|---|
| User Study 1 | Accuracy: 100%(11/11)<br>Effective use of tooltips<br>Used both GUI and code editor | Task 3.2: Setting Vehicle Type to the highest level:<br>Took some time to find the field |
| User Study 2 | Accuracy: 91%(10/11)<br>Used search functionality<br>Used both GUI and code editor | Task 4.2: Adding new property:<br>Could not find where to add new property in Schema Editor<br>Did not know how to edit property name |
| User Study 3 | Accuracy: 82%(9/11)<br>Effective use of tooltips<br>Not familiar with JSON schema<br>Used only code editor in beginning and then only GUI editor after task 2.3. | Task 2.3: Duration of Simulation:<br>Did not think about using the GUI panel to retrieve information<br>Task 2.4: Validity of humidity:<br>Mistakenly thought humidity value was valid.<br>Did not consider red underline as an error<br>Task 4.2: Add new property:<br>Set property name in wrong place |
| User Study 4 | Accuracy: 82%(9/11)<br>Solved tasks in a short time<br>Used only the GUI editor | Task 2.4: Validity of humidity:<br>Mistakenly thought humidity value was valid<br>Did not find out that he could use tooltips<br>Task 3.2: Setting Vehicle Type to the highest level:<br>Did not scroll down in Dropdown menu<br>Task 3.3: Validation Errors:<br>Thought red underline was spell checking |
| User Study 5 | Accuracy: 100%(11/11)<br>Solved tasks in a short time<br>Used both GUI and code editor | Task 3.2: Setting Vehicle Type to the highest level:<br>Took some time to find the vehicle type<br>Task 4.2: Adding new property:<br>Did not set the property type at the beginning |

Set the $id field of this schema to : “https://www.example.com/self-driving-vehicle”.

2) The simulation software will get a premium version in the next update, for which the user needs a license. The license key should be stored in the configuration file. Add a new property “LicenseKey” to the “properties” object. It should be of the type “string”. The length of the key is at most 20 characters long. Add a short exemplary description.
3) Go back to the file editor and verify that the new property “LicenseKey” is displayed and that the correct information is displayed when hovering over the property.

### *B. User Study results*

TABLE VI
USER STUDY 1 - FEEDBACK AND SOLUTION

| Feedback | Solution |
|---|---|
| The property value should not be autocorrected if the user enters an incorrect value. Instead, an error message or another way should be used to inform the user that their input is incorrect. | Instead of autocorrecting values, we now provide more clear user feedback on incorrect values (red underline, error symbol). |
| It would be good to have the ability to remove data entries with the GUI panel. | Implemented by adding a *remove* button next to properties which have data and are not required. |
| A search functionality to locate properties would be helpful, especially within nested levels. | Implemented in the toolbar. All findings are highlighted in the GUI panel. |
| The GUI panel feels overwhelming to the user due to many variations in styling and color of the GUI elements. | We slightly reduced the number of different styling by no longer showing required properties in bold face and instead just show an asterisk next to it. |
| The cursor should not have the clickable animation when hovering over non-clickable fields in the GUI editor. | Now we only show the clickable animation when hovering over clickable GUI components. |
| In drop-down menus we do not need a button to clear the selection. | We disabled the option of clearing the selection. |
| If the type of a property is "any", it should not be interpreted as the "string" type in the GUI panel. | We show a drop-down menu to the user, where they can select the type they want to use. |
| Validation errors should not be highlighted via a warning symbol, but instead an error symbol. | We changed the warning symbol into an error symbol. |
| After performing an undo or redo action, the cursor should jump to the corresponding location to reflect the changes made by the user. | Will be considered in future work. |

TABLE VII
USER STUDY 2 - FEEDBACK AND SOLUTION

| Feedback | Solution |
|---|---|
| A graph-based view would be more intuitive for handling complex data structures. | Will be considered in future work. |
| Providing immediate feedback to users when they enter incorrect ranges is essential to prevent them from inputting invalid values into the property. | We now highlight schema violations by a red error symbol in the GUI panel and underlining the property name in red. Additionally, the tooltip lists all schema violations of a property. |
| Validation errors should also be reflected in the GUI panel, including for child properties. | See point above. Also, now the tooltip lists schema violations of child properties. |
| When dealing with an array, the display name of array elements (index) should be improved. Currently, the tool only shows just the element index. | We replaced the numerical labels by a standard programming notation, which is `propertyName[0]`, `propertyName[1]`, ... |
| The input field next to the *Add Item* button is confusing. Both the input field and the button can be used to create a new item, which is redundant. | We removed the input field next to the button. |
| It would be more consistent, if all user input in the GUI panel was within the right column of the table. That in some scenarios user input is needed within the left column (for names of new properties) feels inconsistent. | Because of the nature of JSON schema, we retained the property name within the left column. To make it clear to the user that the property name can be edited, we added an *edit* icon next to it. |
| The search function for locating specific properties lacks clarity at first glance. It should provide an immediate response and extend to nested levels, rather than merely highlighting the higher-level findings. | The search now provides a list of results, and upon clicking on a particular result, it jumps to that result in the code panel and GUI panel. In the GUI panel, if the element is nested, its parents will be automatically expanded. |

TABLE VIII
USER STUDY 3 - FEEDBACK AND SOLUTION

| Feedback | Solution |
|---|---|
| Working with the schema editor is difficult for me. It does not feel intuitive. | We made the schema editor more intuitive by creating our own simplified JSON schema meta schema. For example, advanced JSON schema features are separated from the simple ones. See section V-E |

TABLE IX
USER STUDY 4 - FEEDBACK AND SOLUTION

| Feedback | Solution |
|---|---|
| Modifying or renaming a new property in the GUI panel does not appear to take effect when double-clicking on it. | Renaming properties in the GUI panel can now be done using the *edit* button next to the property name. |
| When creating a new property in the schema editor, its sub-schema has to be selected, such as *string property* or *boolean property*. Additionally, the type of the property has to be selected by the user too. Therefore, for example, when creating a new *string property*, the user has to select that it is a string two times. It would be much more intuitive if the selection needs to be done only one time. | We completely overhauled our JSON schema meta schema. Now, when creating a new property, the user will have to select the type only once. |
| A toggle button should be implemented to enable and disable the code panel and GUI panel. | Only having a GUI panel or only having a code panel restricts the user unnecessarily. The interplay of both panels is what makes this tool most effective. If the user does not want to use one of the panels, they can resize that panel to a very small size. |
| When working with a particular property in the GUI panel the opacity of the other properties should be decreased, visually highlighting the property that currently is in focus. | Will be considered in future work. |
| Simplify the schema editor to make it easier to work with, for those who are not very familiar with JSON schema. | Has been done, see section V-E. |

TABLE X
USER STUDY 5 - FEEDBACK AND SOLUTION

| Feedback | Solution |
|---|---|
| The search button is not immediately evident, making it challenging for users to locate the search function. | Instead of showing the search bar only when clicking the search button, we now always show it. |