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

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, 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 introduce related work and existing schema formats, as well as schema-to-gui approaches. In section III we evaluate existing schema languages to find the one most suitable for this work. Section IV describes the design and architecture of *MetaConfigurator*. Next, in section V, we cover the implementation of *MetaConfigurator*. We conducted a user study, which is described in section VI. Finally, we conclude our work in section VIII.

II. Related Work

This sections covers existing schema languages and existing approaches to generate UIs from them. As our research is of practical nature, we also consider gray literature like 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 in structured data formats.

The following sections describe existing schema languages.

1) JSON schema: JSON is a common data-interchange format for exchanging data with webservices, but also for storing documents in noSQL databases like MongoDB. Because of the popularity of JSON, there is also a demand for a schema language for JSON. One such language is JSON schema [1], [2]. Figure **??** shows an example of a JSON schema and Figure II-A1 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 [3]. Schemas for many popular configuration file types exist. *JSON schema store* [4] 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. [5], [6] give further examples of JSON schema used in practice.

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

2) XSD and DTD: For XML the two de-facto standard schema languages are Document Type Definition (DTD) [7] and XML Schema Definition (XSD) [8]. XSD is the newer

```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 **??**

and more expressive format of them and in large parts replaces and supersedes the more limited format DTD [9]. It is recommended by W3C [8].

Multiple more schema languages have been proposed and developed but are relatively unknown compared to XSD [10], [11].

3) Other schema languages: We consider also other schema languages that are not as popular as JSON schema, XSD or DTD:

(a) CUE (Configure, Unify, Execute) [12] 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 [13] 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) [14] is a schema language for JSON documents, which is significantly simpler than JSON schema.
(d) Type Schema [15] is a schema language for JSON documents, similar to JSON Type Definition but using a different syntax.

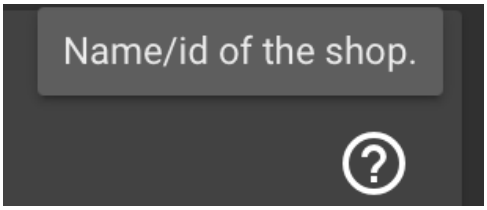

Fig. 1. Tooltip Assistance

(e) GraphQL schema language [16] is a schema language for GraphQL APIs.
(f) Protocol Buffers [17] is a language for data serialization by Google.

We do not consider any graphical modeling languages like 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 like OWL or RDF Schema as they are not intended for data validation but rather for knowledge representation. Future work could investigate if such languages can also be used 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 [18]–[20]. 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 like 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 like auto-completion, syntax highlighting and tooltips. However, they 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 (Figure 1), Auto-Completion (Figure 2) and Choice Selections (Figure 3). By inherently adhering to the schema structure, with such GUIs configuration errors are avoided or at least significantly reduced. Users that 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.

There exist various approaches that generate web forms from a schema, for different frontend frameworks, e.g., *React*

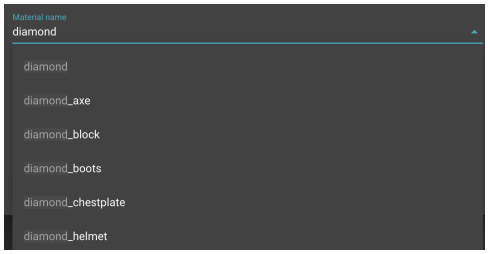

Fig. 2. Auto-Completion

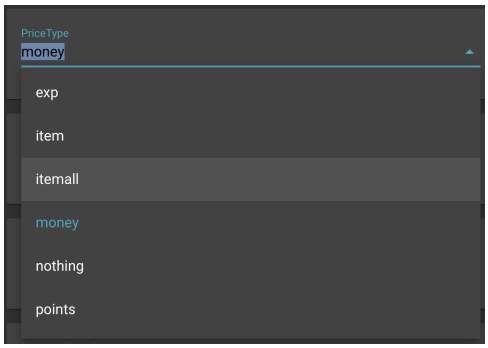

Fig. 3. Choice Selection

JSON Schema Form [21], *Angular Schema Form* [22], *Vue Form Generator* [23], *JSON Forms* [24], *JSON Editor* [**?**], and *JSON Form* [25]. 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 4 shows an example for a generated form using JSON Forms.

The key differences to our approach are that those editors only provide the GUI for editing the data, but not a text-based editor. Also, they do not provide an editor for the schema itself. Finally, all except the last two of the given approaches 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 add need to be created and maintained by the schema author. Our approach does only require the JSON schema and does not require any additional configuration, so the user can use it with any schema file, e.g., from the JSON schema store.

Adamant [26] 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 useful for scientific data. Figure 5 shows an example in the schema edit mode. Adamant differs to *MetaConfigurator* in that it does not provide a text-based editor for the schema and that it is specifically designed for scientific data, while our approach is more general.

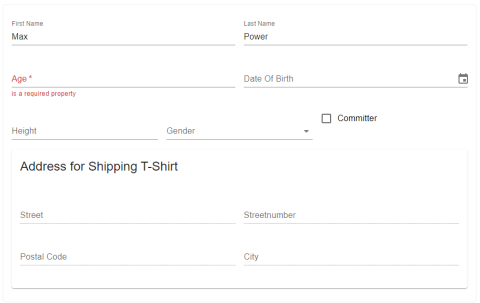

Fig. 4. JSON Forms, example for a generated form

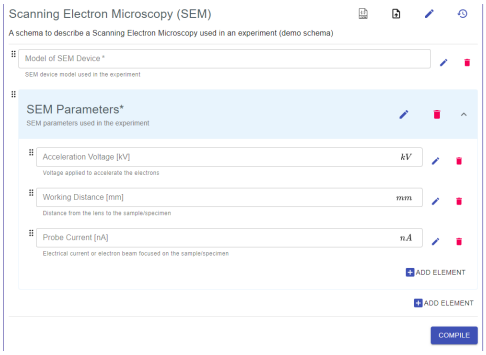

Fig. 5. Adamant, example for a form in edit mode

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).

The closest to our approach is the *JSON Editor Online* [27], which is a web-based editor for JSON schema. 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 6). 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 7). 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 variety of schema editors that are paid software, such as *Altova XMLSpy* [28], *Liquid Studio* [29], *XML ValidatorBuddy* [30], *JSONBuddy* [31], *XMLBlueprint* [32], and *Oxygen XML Editor* [33]. Those are editors for XML or JSON schema, mostly with a combination of text-based and graphical views. In contrast to our approach, they are not web-based and do not focus on editing a JSON document based on a schema, but rather on editing the schema itself. In this regard, they are more specialized than our approach.

3) Schema visualization: Generating a GUI from a schema is related to schema visualization, for which several techniques exist [34]–[37]. However, with schema visualization the focus is only 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.

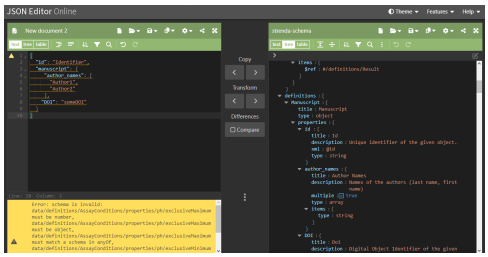

Fig. 6. JSON Editor Online

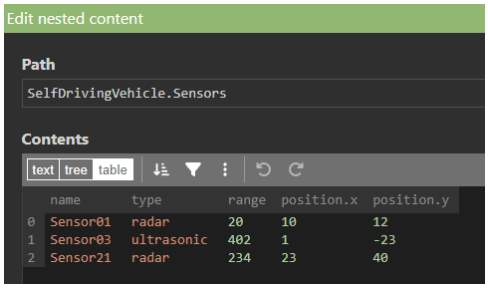

Fig. 7. JSON Editor Online

III. Evaluation of Schema Languages

We evaluate several schema languages in how well they fit our use-case. We consider all schema languages mentioned in section II-A for this evaluation.

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 evaluated the seven schema languages we mentioned in section II-A based on 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 [38] 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

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

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.

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

Table I shows 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. [3], who also found over 80.000 JSON schema files on GitHub, and with the claim of the JSON schema website [2] that JSON schema is the de-facto standard for JSON schema languages.

IV. DESIGN

A. Overview

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.

Note that the design of *MetaConfigurator* is strongly inspired by another tool by one of the authors [39].

Figure 8 provides a sketch of the intended user interface design, showing the *file editor mode*.

The application is divided into two main panels: the *Text editor panel* (on the left) and the *GUI editor panel* (on the right). In the Text editor panel the user can modify their data by hand, just like in a regular text editor. On top of that, features like syntax highlighting and schema validation assist the user. In the GUI editor panel, the user can modify their data with the help of a GUI. This GUI is based on the schema which the user provides (more on that in the next paragraphs). For enum properties, the user will get a dropdown menu with the different options to choose from, for boolean properties the user will get a checkbox and so on. Other features, such as tooltips that display the description and constraints of a property, further assist the user.

This design combines the benefits of both a rich text 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).

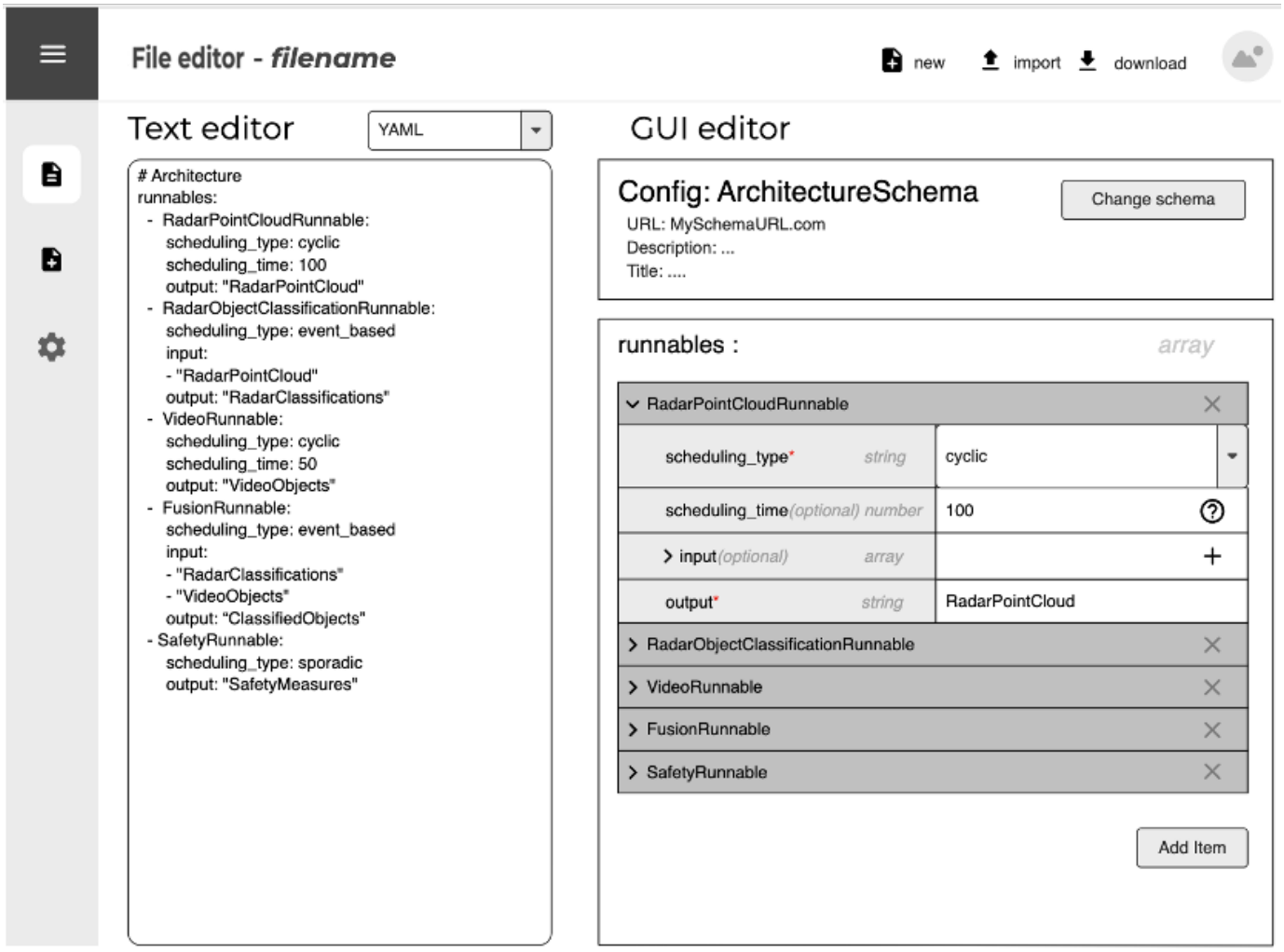

Fig. 8. Sketch of the Tool. File Editor view.

Besides editing their data with the tool, the user can also use it to modify their schema or create a new schema. This is illustrated in figure 9, which sketches the *schema editor mode*. The structure of the user interface remains the same as in the file editor mode. The main difference in this mode is, that the schema being used for the GUI and schema validation is not a user schema, but instead the JSON schema meta schema itself.

The exemplary file editor tab sketch in figure 8 shows user data based on a schema called *ArchitectureSchema*. This *ArchitectureSchema* is shown from the schema editor tab perspective in figure 9.

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 modes, file data and schema being used by the tool differ.

TABLE III
FILE DATA AND SCHEMA FOR THE DIFFERENT MODES

Mode	Effective File Data	Effective Schema
File editor	User data	User schema
Schema editor	User schema	JSON Meta Schema
Settings	Settings data	Settings schema

B. Architecture

The core of our tool 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 rich text 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

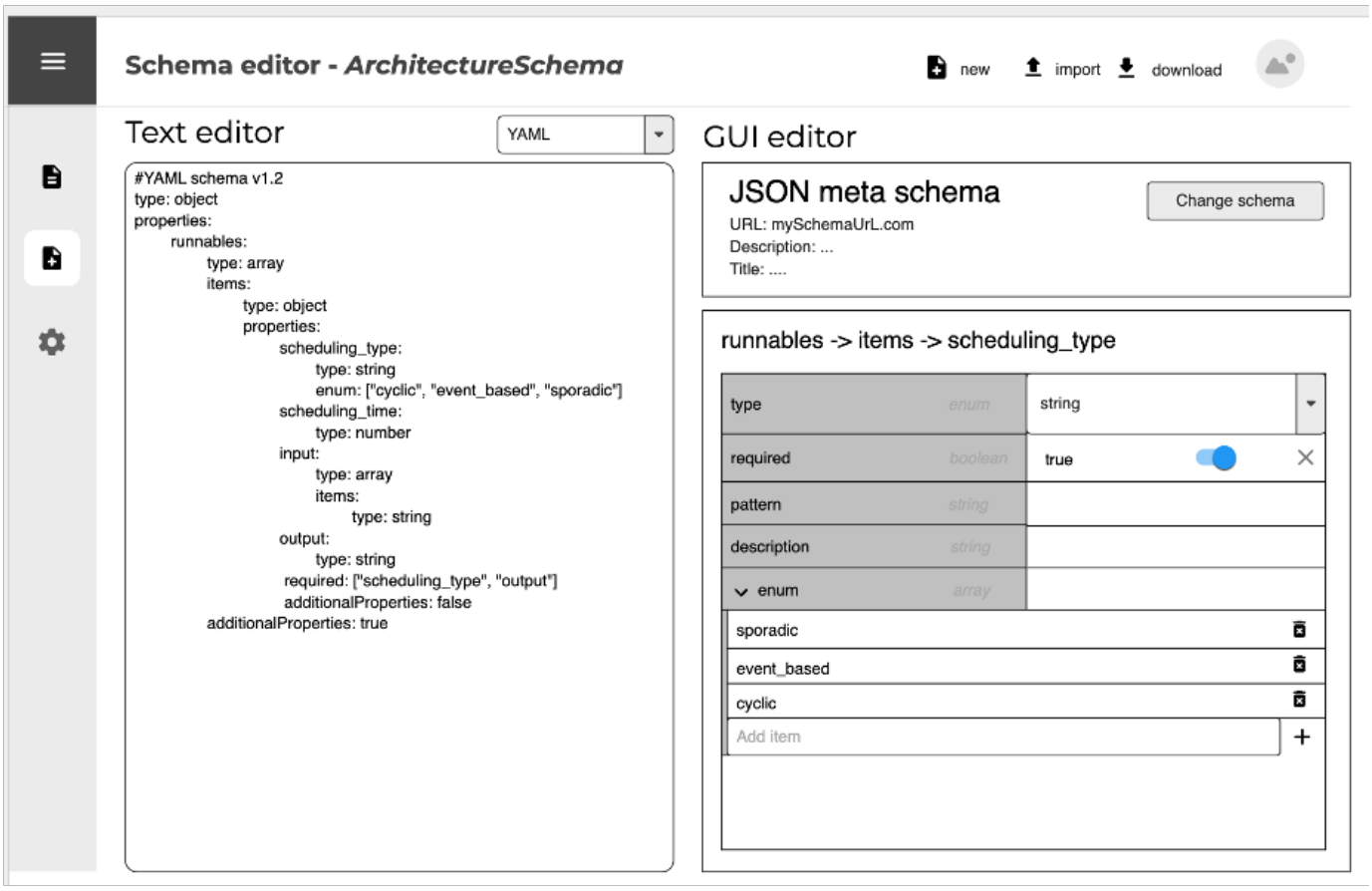

Fig. 9. Sketch of the Tool. Schema Editor view.

is done when the user modifies the data in the editor panel. The following example use-cases illustrate the capabilities of this architecture:

- Format converter: one panel shows the data in a rich-text editor in JSON format, a second panel shows the data in a rich-text 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 rich-text 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 GUI editor panel would require some data schema to know which GUI elements to show.
- 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.

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) Text Editor Panel: For the text editor panels, we embed a rich-text 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 architecture allows for having one text editor panel that supports multiple languages, as well as for having separate text editor panels, one for each language. The panels subscribe to the data store. Whenever the configuration data is changed in the store, the panels will take the new configuration data JavaScript Object, serialize it into the given language 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. An alternative to replacing the complete text in the text editor, whenever data in the store is changed, would be to only replace the section of the text, which corresponds to the change. This would require on the one hand identifying which part of the data is affected by the change and on the other hand understanding of the data within the serialized text in a way that it can be manipulated (e.g. navigating within the hierarchical data structure and changing values of a given path). This is out of scope for the project, but may be part of future work. Even such deep understanding of the text would wipe out non-default formatting at the sections affected by change. We accept that modifying data via the GUI panel will wipe out non-default formatting and also erase comments from

```
editorContent =
  "{
     'people':
       [
         {
           'age': 22,
           'name': 'alex'
         },
         {
           'age': 5,
           'name': 'bola'
         }
       ]
   }";

resultRow1 = determineRow(
        editorContent,
        "people"
 );
 # resultRow1 = 1

resultRow2 = determineRow(
        editorContent,
        "people.0.age"
 );
 # resultRow2 = 4

resultRow3 = determineRow(
        editorContent,
        "people.1.name"
 );
 # resultRow3 = 9
```

Listing 3. Example input and output for *determineRow(editorContent, dataPath)*

YAML documents.

In the future, those difficulties could be tackled in the following way: 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). Whenever the configuration data JavaScript Object is serialized into text, we apply those formatting style settings. Second, 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 [39].

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.

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. See listing IV-B2 for an example input and output of such a function.

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.

For the tool to support a data format, such as JSON, YAML or XML, the following needs to be implemented for that format:

- Function to stringify a JavaScript object to a string in the data format
- Function to parse a string in that data format as a JavaScript object
- determineRow(editorContent, dataPath)
- determinePath(editorContent, cursorPosition)

3) GUI Assistance Panel: The GUI assistance panel(s) directly work with the given schema and provide 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, ...) or complex (array or dictionary of data fields). 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.

C. Intended Workflow

In this section, we outline the workflow of our tool.

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.

V. Implementation

We use vue.js (TODO citation) as the UI framework for our tool, combined with PrimeVue (TODO citation) as the UI component library and Tailwind CSS (TODO citation) for CSS utility. A detailed list of all libraries used can be found in the wiki of our GitHub repository (TODO link). Table IV shows an overview of the most important libraries used and their purpose.

A. Tool Overview

The tool provides a solution for managing configuration files with three distinct views:

- **File Editor** (Figure 10): For editing configuration files, based on the schema provided to the tool.
- **Schema Editor** (Figure 11): For editing the schema, which is used by the File Editor.
- **Settings** (Figure 12): To configure various tool settings.

The user interface of *MetaConfigurator* is structured in the following way (see the numbers in red in figure 10):

1) Button to switch to another view (e.g., from File Editor to Schema Editor).
2) Toolbar with various functionality.
3) Code editor panel.
4) GUI editor panel.

1) Toolbar functionalities:

2) Settings: In the Settings view (figure 12), the user can adjust various parameters (tab V)of *MetaConfigurator*:

B. Code Panel

The code editor is a GUI panel designed for editing the configuration files. For this project, we use the popular *Ace Editor* [40] library to embed an interactive code editor into our user interface.

1) Features: To make our code editor more user-friendly, we implemented several features, which are described in the following.

a) 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* [41] 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.

b) Syntax Highlighting: Ace Editor supports syntax highlighting for both JSON and YAML, which we use. The data format can be switched in the *Settings* page. Note that Ace Editor allows to collapse/expand segments of the text, based on the hierarchical tree structure of the data.

c) Linkage of text with the data model: As described in section IV-B2, 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 every data format to be supported, those two functions need to be implemented.

For *JSON*, the functions have been implemented using a *Concrete Syntax Tree (CST)*. The text content is parsed as a CST. Then the tree is 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, the reverse is done: the CST is traversed until the effective path of the $currentNode$ is the target path. Then $currentNode.range.start$ is returned as 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.

d) Editor Operations: The code editor has more functionalities, such as the possibility to open a file by drag and drop into the editor, as well as customizable font sizes.

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

TABLE IV
Libraries used in the implementation of our tool

Library	Purpose
vue.js	UI framework
PrimeVue	UI component library
Tailwind CSS	CSS utility
@fortawesome/fontawesome-svg-core	Font Awesome SVG icons.
ajv@8.12.0	JSON schema validator for Node.js and browsers.
brace@0.11.1	Browser-based code editor with syntax highlighting and code folding.
primeicons@6.0.1	It is a popular UI component library for JavaServer Faces (JSF) applications.

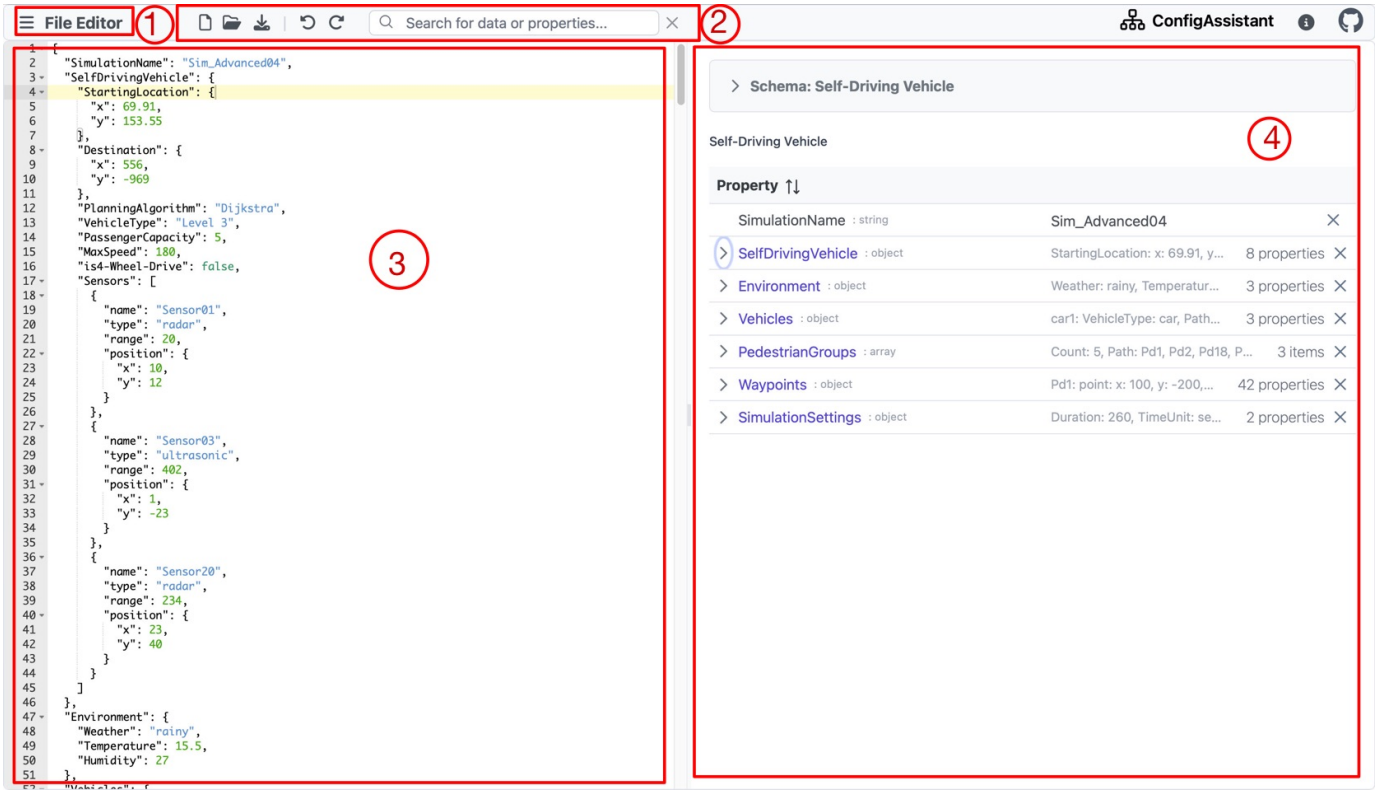

Fig. 10. FileEditor

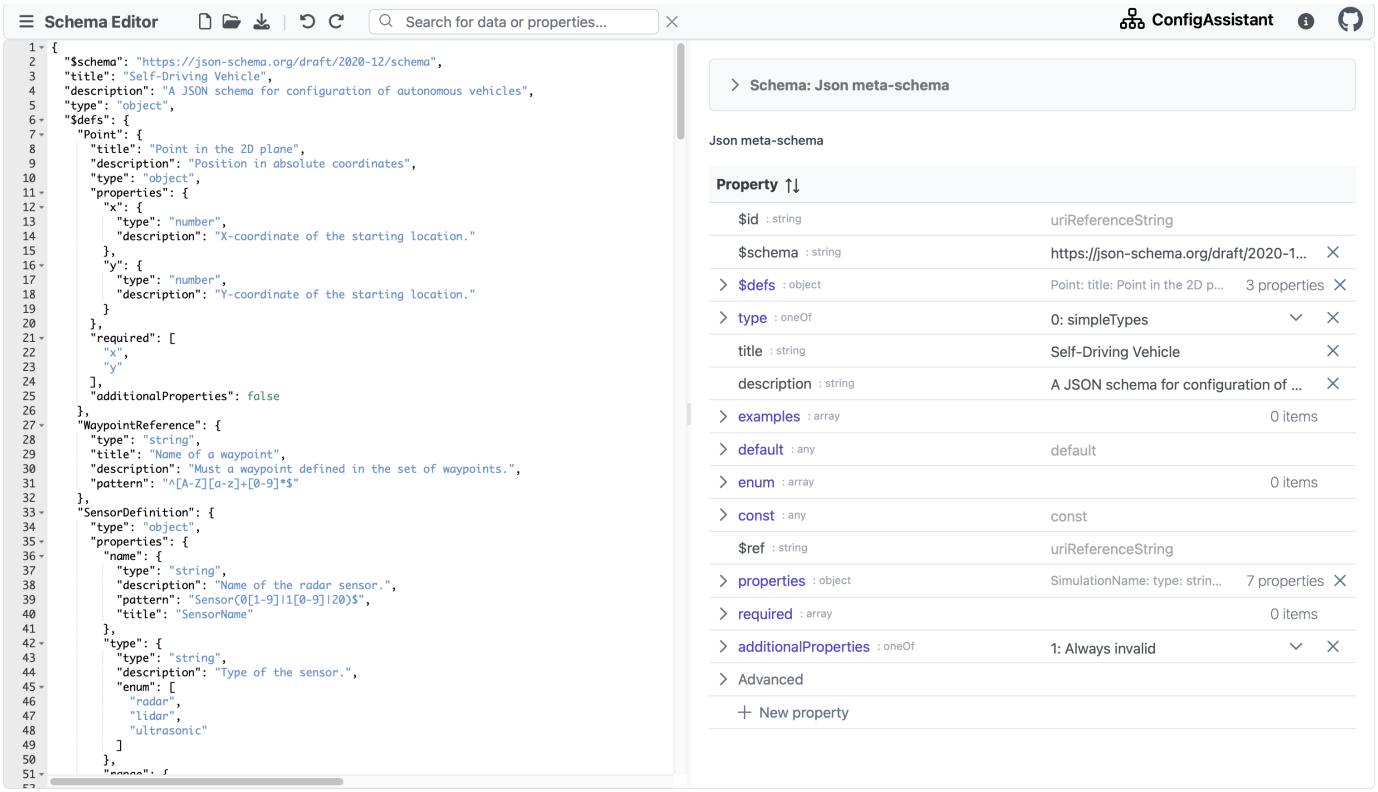

Fig. 11. SchemaEditor

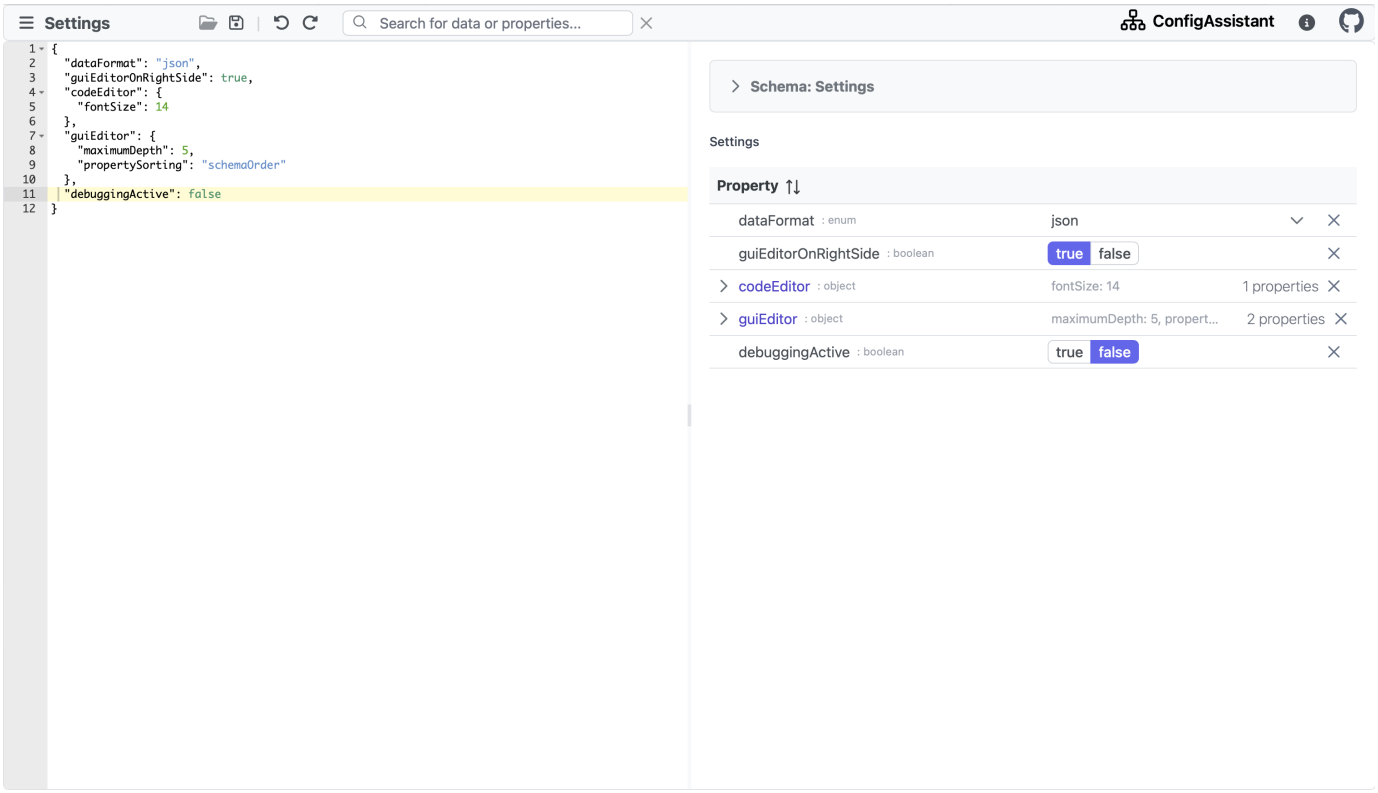

Fig. 12. Settings

TABLE V
SETTING PARAMETERS

Parameter	Description
Data Format	The data format used by the code editor. Currently YAML and JSON are supported.
Code Editor Font Size	Size of the font in the code editor.
Gui Editor On Right Side	By default, the GUI panel is located on the right side and the code panel on the left side. Using this option, their positions can be swapped.
GUI Editor Property Sorting	The order in which the properties in the GUI panel should appear: • *Schema Order*: Orders elements according to the order in the schema. • *Priority Order*: Orders elements based on their priority (e.g. required properties first, deprecated properties last). • *Data Order* : Orders elements the same way as it is in the data.
GUI Editor Maximum Depth	Determines the maximum level of nesting or hierarchy displayed in the GUI editor.
Debugging Active	For debugging purposes, users can enable or disable the debugging panel as needed.

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. Arrays elements are represented similarly, where the index of the array element is the key and the value is the array element itself. Figure **??** shows the GUI editor component with an example schema and configuration data.

To allow this representation of the schema, we do some pre-processing of the schema, which is described in section V-D.

1) Features: To assist the user in editing the configuration data, the GUI editor offers a set of features, which are described in the following.

a) 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. 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* to 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.

b) Type specific components:

c) Components for other JSON Schema features:

d) Resolving data-dependent keywords:

e) 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.

f) Schema Information Tooltip: When the user hovers over the property key or array index, an overlay is displayed, 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 title and description of the property, the overlay then shows constraints (such as the number must be greater than 0) and in 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.

g) Highlighting Schema Validation Errors: When the configuration data does not comply to the schema, the corresponding elements are underlined in red and highlighted with a red error icon. This way, the user knows where any errors are. Additionally, the schema information tooltip lists all schema violations.

D. Schema preprocessing

To represent the schema in the GUI editor, we preprocess the schema. 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.

1) One-time Preprocessing Step: When the schema is loaded, we perform a one-time preprocessing step that currently only involves migrating the schema to the newest version, as described in section V-F. The user will be informed about this step and also prompted with a dialog, when the schema file does not define which JSON schema version it uses. After the migration, the resulting schema is loaded into the tool in the *Schema Editor* page.

2) Internal preprocessing: This preprocessing steps are mainly used to generate the GUI editor and thus are internal steps that will not be visible for the user. They happen at every layer of the schema tree lazily, only when required. 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 details.

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 V-D2a shows an example schema, Listing V-D2a shows the equivalent example after this first preprocessing step.

b) Resolving allOfs: The `allOf` keyword in JSON schema specifies that all of 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 to 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* library for this task. Listing V-D2a 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 support for allOf and any other keywords for which we use this in the preprocessing is limited.

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

Listing 4. Simple JSON schema before reference resolving

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

Listing 5. Simple JSON schema after reference resolving

c) Converting types to oneOf: In JSON schema, a property can have multiple supported types, such as shown in listing V-D2c. A semantically equivalent schema can be generated by the use of oneOf, where each sub-schema contains exactly one of the types, as shown in listing V-D2c. If a schema defines more than one type, we convert the types to oneOf. This way,

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

Listing 6. Simple JSON schema after allOf resolving

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

Listing 7. Simple JSON schema with two possible types

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

Listing 8. Simple JSON schema after conversion of types to oneOf

we reduce complexity by having to solve only the problem of oneOf. For schemas that already contain oneOf, every type is multiplied with every existing oneOf sub-schema. For 2 types and 3 oneOf sub-schemas, this will result in a new oneOf with 6 options. An exception is when a type can not be merged with an 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.

d) Removing incompatible oneOfs and anyOfs: It can happen that a schema has oneOf or anyOf options which 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 V-D2d provides an example of a schema with an incompatible oneOf option. An example where this occurs is in our adjusted JSON schema meta schema, if we assign the root property the jsonSchema sub-schema, which allows objects and booleans, but then additionally restrict the root to be an object. 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 (can not be merged) are removed (see listing V-D2d).

e) Merging singular oneOfs and anyOfs: Because of the previous pre-processing step, it can happen that for some oneOfs or anyOfs there remains only one compatible sub-schema left (see listing V-D2d). 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 V-D2e).

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

Listing 9. Simple JSON schema with incompatible oneOf option

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

Listing 10. Simple JSON schema with incompatible oneOf option removed

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

Listing 11. 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 all types (when there is more than one type) 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 an 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 oneOfs into the anyOfs: 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.

h) Title inducing: The `title` keyword is used to give a schema a short description. This is not necessarily the same as the property name of properties of an object. As we use the title in various cases to display for the user, we inject the property name in cases where no explicit title is given.

i) Processing enum and const: The enum keyword is used to restrict the values of a field to a fixed set of valid values. The const keyword, similarly, restricts the property value to a single allowed value. Thus, setting the const value is equivalent to settings the enum value with an array that contains this single value.

We convert any usage of const to enums with a single element, which allows us to ignore the const keyword in other operations.

```
{
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    }
  }
}
```

Listing 12. Simple JSON schema with one property without a title

```
{
  "type": "object",
  "properties": {
    "name": {
      "title": "name",
      "type": "string"
    }
  }
}
```

Listing 13. The property names was used for the title field

3) Calculating an effective schema: This third preprocessing step is calculated the most often, namely every time the data changes. However, for most schemas this preprocessing step is trivial. 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.

E. Schema editor

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 (TODO citation) 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

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

Listing 14. 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.

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 (TODO citation) 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 `$vocabulary` keyword yet. Both features are used in the JSON schema meta schema, as it is distributed over multiple schema files. To circumvent that problem, we put all schemas in one schema file into the `$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.

Thus, we added `if` conditions to each field to only show them when they make sense in the current context. Listing V-E4 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. [3], which analyzed the usage of JSON schema keywords in 82,000 JSON schemas.

F. JSON schema versions

JSON schema has had 10 different drafts over the years, the newest being draft 2020–12 [2]. In real-world we cannot expect all schemas to have the newest version. Baazizi et al. [3] investigated over 82.000 open-source schemas in 2021, where they found that most of them are using draft 4, which was released in 2013. As the different drafts are not necessarily compatible with each other, tools supporting one draft become outdated when a new draft releases. However, Viotto et al. [42] provide a library for migration schemas from older versions to the newest draft without loss of information. Thus, by using this library, our tool only needs to support the newest draft directly. If a user uses a schema from an older draft, we first migrate it to the newest draft internally.

VI. User Study

We conduct several interviews with professors and students as potential customers. 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 [**?**], provided by professors that we worked with) 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, which could be found in the appendix. A-A 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.

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

We performed the user study on 5 participants.

*1) **RQ1:** Which aspects of the tool can be improved?:* Tables VI-XI 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 VII shows the accuracy and difficulties that the participants had when solving the tasks.

TABLE VI
USER STUDY 1 - FEEDBACK AND RESOLUTION

Feedback	Resolution
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.

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?:*

VII. DISCUSSION

A. Implications of our work

B. Threats to validity

C. Future work

VIII. CONCLUSION

We developed a tool (*MetaConfigurator*) that generates YAML/JSON file editor GUIs tailored to a given data schema.

First, we evaluated existing schema languages by expressiveness and popularity and chose JSON schema as the most suitable for this work. We created a mockup showing how the tool should look like and designed the architecture, with modularity being one of the key aspects. For example, data formats (JSON/YAML/XML) or the different panels (GUI/rich text code editor) can be seen as building blocks that are exchangeable. We implemented *MetaConfigurator* according to that design. One of the main challenges was supporting the different JSON schema features, such as conditions and *oneOf*, *anyOf*, for which we introduced multiple schema preprocessing steps.

We conducted a user study with three professors and two researchers, where they had to solve particular tasks using the tool and then provide feedback about it. All five participants provided useful suggestions for improving the tool, which we then incorporated into the tool. They rated *MetaConfigurator* very positively and answered that they would use it in practice.

Finally, we outlined future work that could be done to further progress *MetaConfigurator*.

ACKNOWLEDGMENTS

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

TABLE VII
RESULTS OF USER STUDY

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

TABLE VIII
USER STUDY 2 - FEEDBACK AND RESOLUTION

Feedback	Resolution
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 IX
USER STUDY 3 - FEEDBACK AND RESOLUTION

Feedback	Resolution
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 X
USER STUDY 4 - FEEDBACK AND RESOLUTION

Feedback	Resolution
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 XI
USER STUDY 5 - FEEDBACK AND RESOLUTION

Feedback	Resolution
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.

REFERENCES

[1] 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

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

[3] 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.

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

[5] 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

[6] 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

[7] 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

[8] 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/

[9] 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

[10] 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

[11] 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

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

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

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

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

[16] 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

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

[18] 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

[19] 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

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

[21] 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].

[22] 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].

[23] 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].

[24] 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].

[25] 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].

[26] 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.

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

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

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

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

[31] "JSON Schema editor for Windows — json-buddy.com," https://www.json-buddy.com/json-schema-editor.htm, [Accessed 08-10-2023].
[32] "XML Editor - XMLBlueprint — xmlblueprint.com," https://www.xmlblueprint.com, [Accessed 08-10-2023].
[33] "The complete solution for XML authoring, development and collaboration. — oxygenxml.com," https://www.oxygenxml.com/xml_developer.html, [Accessed 08-10-2023].
[34] 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
[35] 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
[36] 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
[37] 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.
[38] P. Carbonnelle, "TOP IDE Top Integrated Development Environment index — pypl.github.io," https://pypl.github.io/IDE.html, [Accessed 18-May-2023].
[39] F. Neubauer, "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].
[40] F. J. Harutyun Amirjanyan, "Code editor," Mar 28, 2010, accessed 14-05-2023. [Online]. Available: https://ace.c9.io/
[41] E. P. etc., "A useful library for validating a json file based on the schema," May 17, 2015. [Online]. Available: https://ajv.js.org/
[42] J. C. Viotti and J. Lagoni, "Sourcemeta/alterschema: Convert between json schema specification versions." 2023. [Online]. Available: https://github.com/sourcemeta/alterschema

APPENDIX A
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": "From Example" → "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.
   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.