A Metadata-Based Ecosystem to Improve the FAIRness of Research Software

Patrick Kuckertz[1,*], Jan Göpfert[1,5], Oliver Karras[2], David Neuroth[1], Julian Schönau[1], Rodrigo Pueblas[1], Stephan Ferenz[3], Felix Engel[2], Noah Pflugradt[1], Jann M. Weinand[1], Astrid Nieße[3], Sören Auer[2,4], and Detlef Stolten[1,5]

[1]Forschungszentrum Jülich GmbH, Institute of Energy and Climate Research – Techno-economic Systems Analysis (IEK-3), 52425 Jülich, Germany
[2]TIB - Leibniz Information Centre for Science and Technology, 30167 Hanover, Germany
[3]Department for Computer Science, Carl von Ossietzky University of Oldenburg, 26129 Oldenburg, Germany
[4]University of Hannover, L3S Research Center, 30167 Hannover, Germany
[5]RWTH Aachen University, Chair for Fuel Cells, Faculty of Mechanical Engineering, 52062 Aachen, Germany
[*]Correspondence: p.kuckertz@fz-juelich.de

2023

Abstract

The reuse of research software is central to research efficiency and academic exchange. The application of software enables researchers with varied backgrounds to reproduce, validate, and expand upon study findings. Furthermore, the analysis of open source code aids in the comprehension, comparison, and integration of approaches. Often, however, no further use occurs because relevant software cannot be found or is incompatible with existing research processes. This results in repetitive software development, which impedes the advancement of individual researchers and entire research communities. In this article, the *DataDesc* ecosystem is presented – an approach to describing data models of software interfaces with detailed and machine-actionable metadata. In addition to a specialized metadata schema, an exchange format and support tools for easy collection and the automated publishing of software documentation are introduced. This approach practically increases the FAIRness, i.e., findability, accessibility,

interoperability, and so the reusability of research software, as well as effectively promotes its impact on research.

Keywords: Research Data Management (RDM), FAIR, software metadata, interface description, semantic software description, software publication, software reuse, machine-interpretable, application profile, CodeMeta

1 Introduction

Research in many academic disciplines relies on computational methods, to the degree that the utilization of software has become integral in numerous fields. Thus, the efficient discovery and reuse of research software is essential for academic progress and communication. Furthermore, the examination of open source code aids in the comprehension, comparison, and integration of methodologies, and the application of software enables users with various academic backgrounds to replicate, validate, and build upon study findings. Scientific software publications are also becoming increasingly important for measuring the research impact and so for the reputation of individual researchers [1, 2].

Finding compatible software that meets researchers' content requirements and integrates seamlessly into existing research workflows remains a significant challenge [3]. Currently available software metadata schemas, such as *CodeMeta* [4], only focus on general information and omit detailed technical descriptions of interfaces, which are important for interoperability and subsequent use [5]. At most, such information can be found on software documentation sites, where it is neither standardized nor machine-actionable. Furthermore, metadata is stored and exchanged in various formats, from which no standardized exchange format has yet been developed, that would allow the broad reuse of metadata once it has been captured [6]. Therefore, in order to make a software known on various platforms and increase its impact, metadata must often be repeatedly collected for each platform separately, which greatly increases documentation effort, which is already perceived to be high. At the same time, the broad dissemination of metadata is essential for the long-term discoverability and subsequent use of software [7]. As a result, researchers must invest considerable effort in both documenting and publishing metadata, as well as finding and integrating research software. Every time software is not found and reused and instead redundantly developed, a significant increase in avoidable programming, documentation and maintenance efforts is imposed.

To address these issues, adaptations of the FAIR Guiding Principles, which aim to increase the findability, accessibility, interoperability, and reusability of research data [8], were recently adopted specifically for research software [9, 10]. Amongst other things, these principles require research software to be registered and indexed in searchable platforms, and annotated with rich metadata. In order to increase the interoperability of software components, the metadata must include interface definitions of modular software architectures, making interoperability the most challenging amongst the four high-level principles. On the one hand, all metadata must comply with domain-relevant community

standards in order to be easily understandable for researchers. On the other, the metadata must be machine-actionable for automated software discovery. In practice, however, it is unclear how the postulated abstract principles may be put into action [11].

The DataDesc ecosystem presented in this article is a practical approach to improving the interoperability and findability of research software. It centers around a software metadata schema that describes the data models on which software interfaces are based. In order to capture characteristics that are usually only described in documentation, metadata elements from various established schemas were reused, combined, and supplemented with new ones. In addition, the ecosystem provides an exchange format in which this information is mapped in a machine-actionable manner. The hierarchical data structure of the OpenAPI standard was chosen as its basis to facilitate its reuse in automated processes. Finally, it includes a toolset that makes it easy to capture and publish software metadata from the source code.

The remainder of this article is structured as follows: Section 2 presents a review of existing software description schemas, addressing different formats in the tension between metadata and documentation. Furthermore, automated description tools and software publication platforms are compared on the basis of the metadata formats they generate or use. Section 3 explains the different components of the DataDesc ecosystem. First, the DataDesc schema is described along with the typical data flow between individual interface components of research software on the basis of its contents, formats, value ranges, and structures. Then, an explanation of the structure of the exchange format and the individual tools that support metadata generation is given. Finally, pipelines to publication platforms are described with which the metadata can be disseminated in a partially automated way. In Section 4, the presented approach is exemplarily applied to the Framework for Integrated Energy System Assessment (ETHOS.FINE) [12] modeling framework from the energy domain, whereupon the strengths and limitations of the DataDesc schema in particular are discussed. Section 5 concludes with a summary of the key characteristics of the presented approach and provides an outlook on future work.

2 Related Work

An overview of current software description schemas is provided in Section 2.1. Additionally, software publishing platforms and automated description tools are contrasted in sections 2.2 and 2.3.

2.1 Software Description Schemas

Software Metadata Standards. Metadata schemas (or standards) are sets of metadata elements (or terms) that are compiled to unify the description of artifacts within their scope. Many different metadata schemas exist for a variety of use cases. Whereas *Dublin Core* [13] outlines general metadata terms,

the *DataCite Schema* [14] focuses on describing research data. *schema.org* is intended to describe web pages with structured data markups but it is also widely used for other purposes [15].

With respect to research software, CodeMeta is a popular community-driven metadata standard. It is based on *schema.org*, which it augments with several additional terms. Various crosswalks exist - that is, mappings from one schema to another - between CodeMeta and other metadata schemas. CodeMeta covers many aspects of software metadata, with some terms focusing on technical details such as file size or operating system and others on administrative information like licenses and links to the software repository. It supports the unambiguous assignment of authors, contributors, licenses, and more via Uniform Resource Identifiers (URIs). The purpose of a software can be specified by means of a textual description, application categories, keywords, and a link to a README file or reference publication. Apart from a coarse classification, the declaration of a software's purpose is therefore still far from being readily machine-actionable; that is, without interpreting (or misinterpreting) natural language. Furthermore, CodeMeta does not include terms for specifying the input and output of a software, nor does it include terms for specifying features or methods implemented by a software. Similarly, the *Citation File Format* schema defines general metadata for the citation of software repositories without describing the software's purpose and interface [16].

In the domain of geoscience, Garijo et al. developed the *Software Desciption Ontology* [17] by extending their own approach, namely *OntoSoft* [18]. OntoSoft elements are structured in six categories: identify, understand, execute, do research, get support, and update. The ontology captures technical metadata like programming language and dependencies and descriptive data like name, website, and contributors. The authors added a description of the input and output data also utilizing the *Scientific Variables Ontology* and aligned *OntoSoft* with CodeMeta. The metadata are published to an open knowledge graph [19]. Garijo et al. support the linking to other instances in the semantic web, like Wikidata, the *Scientific Variables Ontology*, and others. Additionally, they developed programs to support researchers in metadata creation and the search for software models [20, 21].

In the domain of bioinformatics, Ison et al. developed the metadata schema *biotoolsXSD* for the software registry *bio.tools* [22, 23, 24]. The metadata is expressed as an XML schema and contains 55 elements, of which ten are mandatory. The use of the EDAM ontology as value vocabulary is required for elements such as function, input, and output. The metadata schema also contains software-specific elements like programming language, license, and operating system. The use of an ontology is not required for these.

Interface Description Standards. In order to increase its technical interoperability and reusability, software can be documented by means of interface description languages. The syntax of such a language enables the formal and programming language-agnostic description of interface functions and their

parameters. Well-known representatives include the Web Service Description Language (WSDL) [25] and Web Application Description Language (WADL) [26]. Both are XML-based specification standards that describe the syntactical elements of web services and, primarily, how to access them. They are utilized to simplify the information exchange in Web 2.0 application development. Whereas WSDL is used in conjunction with SOAP, WADL enables the description of HTTP (and in particular REST-conform) web services. Both languages provide machine-processable descriptions but do not support taxonomy or ontology information for semantic classification. The WSDL and WADL standards were last updated in 2007 and 2009, respectively.

The OpenAPI specification is an interface description language that focuses on REST APIs [27]. By utilizing YAML and JSON, it is both machine-actionable and human-readable. By default, it is used to define the general properties of APIs, such as the version, contact, and license information or server names and addresses. However, it also defines technical aspects, mainly with respect to REST interface functions like the paths to endpoints, HTTP verbs, parameters or response code descriptions. The OpenAPI standard also allows for the annotation of custom properties using a concept called *extensions* or *x-attributes*. These extensions provide a powerful way of describing additional functionality not covered by the standard specification. As an open and non-proprietary state-of-the-art industry standard, the OpenAPI specification is actively maintained and regularly updated.

The Web Ontology Language for Web Services (OWL-S) defines ontologies built on top of the Web Ontology Language (OWL) for describing semantic web services on a technical level, making it more powerful but also more complex than regular description languages (i.a., WSDL and WADL) [28]. It describes the purpose of services, how they are accessed, and how they function. Although more powerful than comparable description languages, OWL-S is not an 'end-all-be-all' solution to service descriptions and requires domain-specific ontologies for describing domain-specific functionality. Furthermore, its focus on semantic web services greatly reduces its legibility; it was last updated in 2004 and is not suited to easy human reading.

The Functional Mock-up Interface (FMI) is an open-source standard for simulation software interfaces [29]. All simulation models whose interfaces have been designed along the standard become so called functional mock-up units (FMUs). The standard ensures that all FMUs are compatible with one another and can be executed in combination on the basis of XML and binary files and C code containing functions, variables, and mathematical formulas. FMI comes with its own documentation standard, namely the FMI Description Schema, which only applies to FMI conform software. It encompasses general information regarding the FMUs such as name, version, author, and license, as well as technical information like model structures, unit, and type definitions. The schema allows structured extensions to the base standard in order to flexibly meet additional requirements. FMI is still actively maintained today and is used in many industrial companies.

Non-standardized Software Description. Software is also described on web pages, where the use of specific terms is typically enforced, but without adopting a metadata schema, thereby only establishing uniformity on the web page itself. Schwarz and Lehnhoff [30], for example, describe a catalog of energy co-simulation components. They use a semantic media wiki to collect information on simulators and add descriptions to the simulation interfaces. The elements of the catalog, which can be used for a metadata schema, are not described in greater detail. The open energy modeling initiative (openmod) includes a list of energy models in their wiki [31]. For each of these, administrative and descriptive metadata are listed, such as license, link to a code repository, model class, and other. The descriptive elements include detailed information on the models. The elements are not formalized as metadata schema and controlled vocabularies are used for neither the elements nor the values. The Open Energy Platform (OEP) introduces framework and model factsheets to describe frameworks and models [32]. These descriptions have been further developed based on the non-formalized openmod metadata elements.

In addition to the description by means of metadata, software is described in documentation and specification websites, providing guidance for both users and developers (e.g., see [33]). The design ideas and specific technical elements of software are typically defined along with their underlying algorithms and procedures. Specifications for the API, user manuals, and examples of applications make it possible to correctly utilize the software. Software documentation is predominantly written in natural language and, therefore, is neither machine-actionable nor easily searchable or comparable. Although such documentation provides rich information, it is not typically considered as part of software metadata.

It should be noted that existing software metadata schemas do not include technical documentation about interfaces. And although interface description languages are designed to collect this information, they focus on web services and protocols. As a result, the interface information of software that is not provided as a service is primarily published as non-standardized and non-machine-actionable information on web pages, often without any connection to controlled vocabularies or ontologies. For research software, most of which is not provided as a service, there is not yet a suitable schema that enables semantic interface descriptions. However, the near-code structures of interface description languages and the ability to connect some via extensions to established software metadata schemas offer promising foundations.

2.2 Software Description Tools

Documentation is generally regarded as an essential component of software development, and yet it is frequently neglected. This is often due to the fact that considerable effort is involved in writing detailed, well-structured, and version-controlled documentation. A recommended means of alleviating this issue is the use of automated documentation tools [34], which are specifically designed to

aid in the process of creating comprehensible and complete documentation for a software project. There are many such tools available, and although the general objective is the same, they differ in their approach, programming language, or input and output formats.

Many of these tools, e.g., Javadoc [35] or Perldoc [36], focus on single programming languages and use source code as their main inputs. By parsing the code, they obtain information on defined types and functions and their relationships. Some documentation tools, such as Doxygen [37] or the Sphinx plugin Napoleon [38], are able to extract this kind of information from bare code; other tools, however, rely on code comments in a determinate format. In either case, additional metadata is typically conveyed via comments. This can comprise, for example, a general description or explanation of a function's parameters. Such information is mostly given as free text and is placed into the final documentation without change. MkDocs [39] and, in some cases, Sphinx [40] constitute an exception by only manually parsing created files, e.g., containing *reStructuredText*. They can, however, both be extended with plugins that automatically generate said text files from code. The output of documentation tools is nicely-formatted documentation pages, typically using HTML or LaTeX. These pages are easily readable and comprehensible to humans, but hardly machine-actionable. Roxygen2 [41] also generates intermediate files that are, in theory, machine-actionable, but, due to their custom data format, are limited in their reusability.

In this regard, Swagger [42] can be distinguished from other tools. Swagger is used primarily for documenting REST APIs and provides a set of distinct but related tools for that. At its core, Swagger utilizes a YAML file standardized in the OpenAPI Specification. This file is machine-actionable and stores all metadata of an API in a structured, hierarchical way. It can be created manually or generated from code, and, when passed to the appropriate Swagger tools, is used to generate a human-readable documentation web page. Unlike many other tools, Swagger does not require specially-formatted comments within the code in order to extract the information. Furthermore, Sphinx can be extended by a plugin to enable support for OpenAPI specification files, which, as implied, makes it possible for Sphinx to generate interface descriptions from OpenAPI compliant YAML.

It should be highlighted that software and, therefore, interface documentation can be parsed automatically from source code and many documentation tools are available. However, most of these rely on code comments that are formulated in natural language and which, therefore, are not directly machine-actionable. In this regard, Swagger is an exception, as it centers around a universal, machine-actionable and standardized metadata file, which is suitable for documentation pages as well as automated reuse. Even though Swagger is intended only for documenting REST interfaces, there is no lock-in to individual programming languages. Because of this inherent flexibility, it offers some potential for the development of generic software documentation workflows.

2.3 Software Publication Platforms

Software can be made discoverable and available for reuse by being published on a variety of software-specialized publication platforms. Thereby, the distinct purposes and objectives of these platforms vary. Although some store the source code of a software in versioned repositories (e.g., [43]), in particular to enable its further development, others aim at the distribution and easy integration of mature programs (e.g., [44]). Some platforms serve as registries, indexing large collections of software and making them searchable using detailed metadata (e.g., [45]). Others are dedicated to the provision of technical documentation and user guides (e.g., [46]).

Furthermore, most of the software publication platforms differ in the data formats they accept and in the uploading processes they provide. Even when using similar file formats, the required information or information structures vary. Some platforms, such as Github [43], Gitlab [47], Bitbucket [48], or Sourceforge [49], ingest the source code directly without a specific required structure. Others support the inclusion of metadata configuration files. For example, Anaconda Distribution [44] requires a YAML file that describes the project. Maven Central [50] requires an XML POM file for storing metadata. Whereas PyPi [45] requires a TOML file with information about packages, NPM [51] generates a JSON file based on text prompts. Swaggerhub [52] requires an OpenAPI-conforming interface description file in YAML format, containing function and argument specifications. Like ReadTheDocs [46], some platforms require a software project to have a documentation folder according to a standard. In this specific case, Sphinx or MkDocs can be used in order to generate such a folder. Platforms like Gitbook [53], CRAN [54], or Github Pages [55] require programming language-specific files for the installation. For example, submitting a project to CRAN requires first creating a TAR.GZ file. Github Pages [55] can store project documentation via HTML files. The OEP [32], Open Research Knowledge Graph (ORKG) [56, 57, 58], or *bio.tools* [23], for example, require manually filling forms with project data in order to register it.

There is no question that publishing platforms are critical to the dissemination, findability, and reusability of research software within and across academic communities. It is advantageous to employ various platforms in parallel to utilize their distinct strengths to increase the impact and transparency of a software. However, as no uniform format for the exchange and subsequent use of software metadata has yet been identified, metadata must often be collected redundantly and adapted to heterogeneous formats and processes, creating the need for a machine-actionable and programming language-agnostic exchange standard.

3 The DataDesc Ecosystem

This section introduces the DataDesc ecosystem. As a central component, the DataDesc schema, which enables the thorough description of software interfaces,

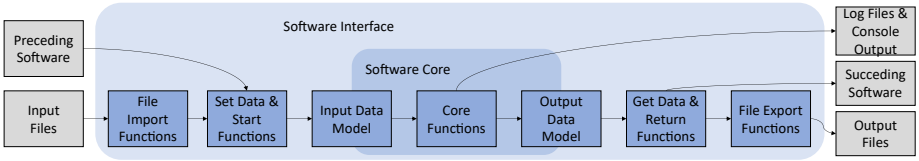

Figure 1: Schematic representation of the generic information flow between software interface and core components.

is explained in Section 3.1. Then, in Section 3.2, an exchange format as well as assistance tools are presented, enabling the gathering, storage, and reuse of machine-actionable metadata. Finally, procedures that can be used to share metadata on publishing platforms are defined in Section 3.3. DataDesc has been released with all of its components presented here under the open MIT license on GitHub [59].

3.1 DataDesc Schema

Metadata schemas often focus on general information provision, which primarily includes the naming of organizations and persons involved in the development process and the technical and licensing conditions under which the software can be obtained and used. By specifying categories and keywords, they also make a valuable contribution to supporting the findability of software. Within these schemas, however, the description of interfaces can only be superficially embedded in general metadata elements. Although this information already provides important insights into a software, it is not sufficient to facilitate its interoperability and reusability in a machine-actionable way. Therefore, the DataDesc schema[1] compensates for this omission by providing a framework for the detailed annotation of individual software interface components, leading to insights into how and with which data and programs a software can be used. The DataDesc schema promotes the reuse and integration of research software and, thereby, is an ideal extension to existing metadata schemas.

An interface, as schematically depicted in Figure 1, serves as a connection point for users and programs to interact with a software. It is composed of the functions through which data can be inputted into and retrieved from the software. These functions are distinguished from the inner functions, which form the logic of the software core. The program core can only be addressed indirectly via the interface, whereby the structures and formats of the information flow are defined by the interface functions and internal data models.

An interface description performed with the DataDesc schema formally identifies the characteristics of an interface according to a collection of metadata elements, detailed in Figure 2, whose meaning and use are explained in the fol-

[1] More precisely, DataDesc's schema is a metadata application profile, as it combines term definitions from existing metadata schemas for a particular purpose. However, as this technical difference is not decisive, the more common term *schema* is used.

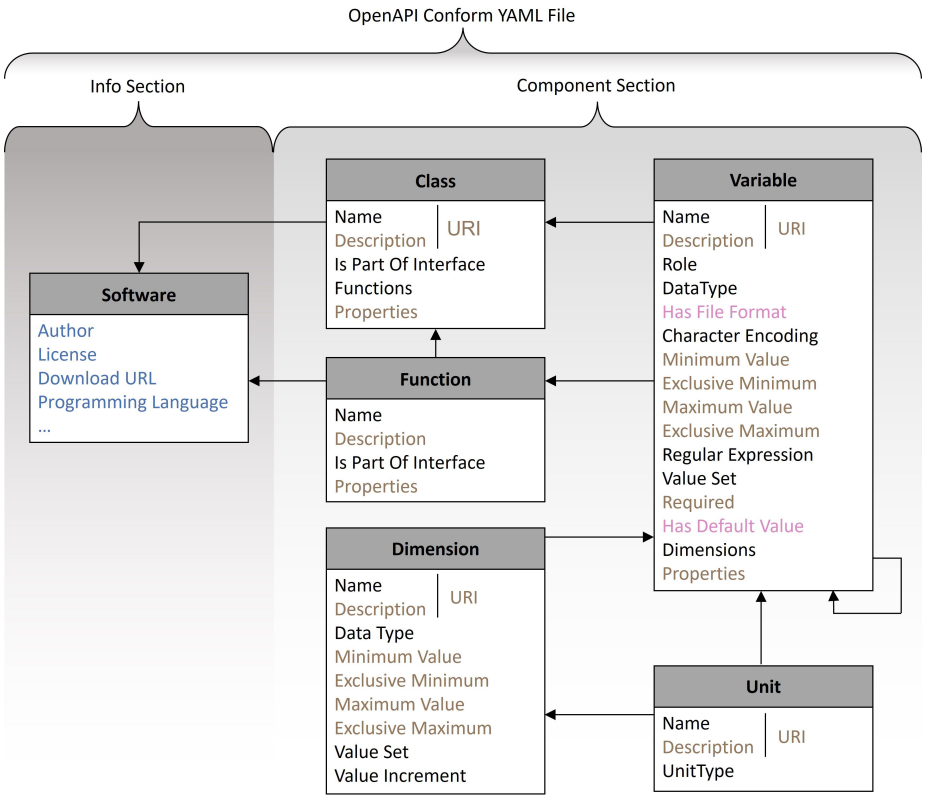

Figure 2: Structure and content of the DataDesc schema within an OpenAPI-conforming YAML file. While a software is described with *CodeMeta* (blue) in the info section, its objects are described in the component section with *DataDesc* (black), which re-uses terms from *OpenAPI* (brown) and the *Software Description Ontology* (pink). All objects are arranged in a hierarchical structure, which is indicated by arrows pointing from child to parent objects.

lowing. The schema comprises the naming (*Name*) and description (*Description*) of all functions, which are part of an interface (*Is Part Of Interface*) and over which a software can be addressed. In order to enable easy and in particular error-free use of a software, the functions' parameters, as well as their underlying data models, must be described in detail. To adequately characterize variables serving as the input or output parameters of interface functions (*Role*), their intended and allowed data must be described in terms of contents, formats, values, and structures.

Data Content Description. In order to digitally process information, it must be stored in the form of variables. In the course of software development, the data content of each variable is defined. This refers explicitly to the referencing

of real world concepts, such as the height or weight of a person, and not of data types, which specify whether variables can contain *integers*, *floats*, *strings*, or similar. A precise understanding of the meaning of the data content a software requires, processes and outputs is essential for its correct and intended use.

However, capturing meanings is not a trivial task. Depending on the demand for precision and generality, describing data content involves varying degrees of effort. The easiest approach is to sensibly name variables during software development (*Name*) and explain them further in docstrings (*Description*). However, these names and free text descriptions almost always leave considerable room for interpretation as to the meaning of the data content. Instead, it is more interoperable to reference concepts from ontologies (with their respective *URIs*), which often provide unambiguous definitions that are agreed upon in the respective research domain [60]. Of course, the open collaborative development of such concepts with the broadest possible participation and agreement within a domain is a labor-intensive process requiring well-organized community infrastructures [61].

If the variable is numerical, documenting the meaning alone is insufficient for fully describing its data content. In this case, additional information about a unit is necessary, so that, for example, a duration of seven hours can be distinguished from one of seven seconds. Just as with the concepts before, a unit can be specified by a name (*Name*) and a description (*Description*), or an ontology reference (*URI*).

In the context of software interfaces, specific concepts and units need not always be declared. In order to enable a greater degree of freedom in data entry and so to enable a more flexible application of a software, intended data contents can be more broadly indicated. For example, specifying the general concept *means of transport* indicates that the software can process data about *bicycles*, *trains*, *cars*, and so forth. Likewise, instead of specific units such as *meters*, *centimeters* or *miles*, the unit type *length* can be used (*Unit Type*). In this context, the use of ontologies offers the advantage that they often already include information that specifically relates to more general concepts.

Data Format Description. The format of a variable defines how the information it contains is to be encoded into binary data and subsequently interpreted. It provides information about which operations may be applied to the data content. The format is defined by the data type of a variable (*Data Type*). There are primitive and complex data types that can be native to programming languages or which come as custom data types provided by libraries. Primitive data types, such as *strings*, *integers*, or *booleans*, can hold single values. Complex data types, like *lists*, *tables*, *arrays*, or *classes*, can group multiple instances of primitive data types.

As complex data types can also recursively contain complex data types, nested structures of arbitrary depth and complexity are possible, although their final level can only contain primitives. Complex structures of this kind are often used to define data models, which summarize the input and output

data of research software into single data objects and make them centrally accessible (e.g., [62], [63]). Oftentimes, classes are used at the highest level for the representation of such data models, to which the interface functions (*Functions*) for importing and storing, as well as for reading out and exporting, are also assigned (cf. Figure 1). In order to describe not only the data types of the function parameters but also the formats nested within them, hierarchical data formats are mapped in the DataDesc schema by means of hierarchical parent-child relationships. To avoid redundant descriptions of complex data models with each function parameter based on them, the DataDesc schema also offers the option of creating separate reusable descriptions of data model classes that can also be referenced (*Name*, *Description*, *URI*, *Is Part Of Interface*). Figure 3 (a.-c.) exemplifies how four variables of the primitive data types of *integer* and *string* are grouped (e.g., in the format of a *class*). Different instantiations of this class are further grouped (e.g., in a *dictionary*).

Files indirectly represent another complex data type, as they can also contain and group data of arbitrary types. As is shown in Figure 1, reading in files is a widely used method of transferring data to a research software, which is why an interface description must also inform regarding permitted file formats that can be processed without errors. For each variable containing a file reference, whatever the variable itself, e.g., file type *string* or *file object*, DataDesc gives the option of specifying the format, e.g., text formats like *XML*, *HTML*, or *TEXT* or binary formats like *PDF*, *XLS*, or *JPG* of a referenced file (*Has File Format*). Beyond that, the character encoding of text formats, e.g., *UTF-8*, *ASCII*, or *ISO 8859-1*, can be specified to support the correct interpretation of text data (*Character Encoding*).

Data Value Description. When creating software, a permissible value range must be defined for each variable, guaranteeing the technically error-free processing and consistency of content for all values from within this range. Technically, the value range is defined in many programming languages by the choice of a variable type. In Java, for example, a variable of type *float* allows all floating point values from $-3.4 * 10^{38}$ to $3.4 * 10^{38}$, whereas a *boolean* can only accept the values of *true* and *false*.

In addition, a value range can be further limited based on content considerations. For example, if a longitude is to be stored in a *float* variable, only floating point values from -180 to $+180$ degrees should be considered valid (*Minimum Value*, *Exclusive Minimum*, *Maximum Value*, and *Exclusive Maximum*). If text variables are only allowed to contain certain patterns, this can be defined through *regular expressions* (*Regular Expression*). For example, if a filename is to consist of only letters, numbers, and underscores, this can be defined using the expression, `^[A-Za-z0-9_]+$`. If the allowed value range of a variable should be fully restricted to predefined values, e.g., *North*, *East*, *South*, and *West*, DataDesc schema offers the possibility of documenting them in the form of value sets (*Value Set*).

It is also part of the value description to specify whether a variable is an

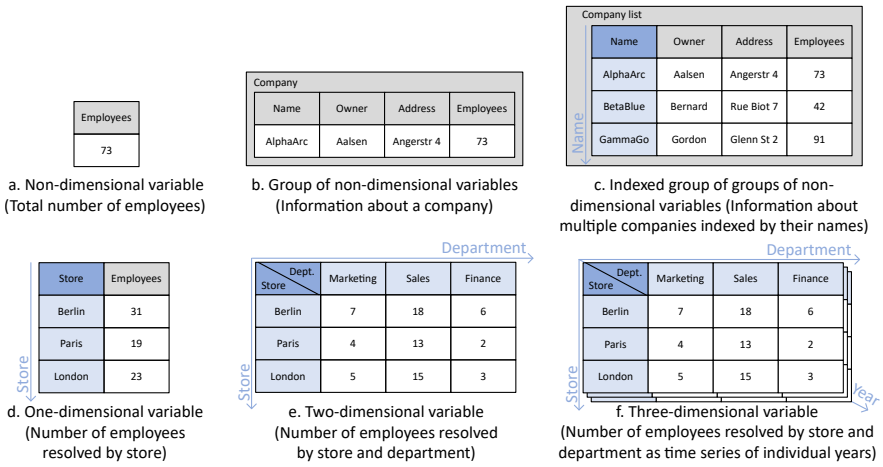

Figure 3: Comparison of widely-used data structures based on an example of information about companies. Variables are shown in gray and their values in white, whereas dimensions are displayed in dark blue with their indices in light blue.

optional parameter (*Required*). If this is the case, it does not need to be set when the respective function is called upon. In this context, a default value can also be specified, and is used if the variable is not explicitly set (*Has Default Value*).

Data Structure Description. For variables of complex data types, the internal data structures must be described at both a technical and contextual level so as to enable the correct accessing of individual values and the determination of their respective meanings. The processing procedures of software programs are designed on the basis of specific data structures whose declaration is the task of interface descriptions. The correct function of a software is not ensured if the structure of the passed data differs from the intended data structure.

Figure 3 (b.) shows four independent variables which, as they represent information characterizing the same single object, are combined in a grouping variable *company*, which itself must be described. The technical structuring of the data thereby maps their context and relates the four variables to each other and to the grouping variable: *Number of employees* becomes *Number of employees of the company AlphaArc*. In order to capture this kind of context within an interface description, the DataDesc schema utilizes hierarchical parent-child structures to map relationships between variables. For this purpose, variables that are the child objects of data model classes or grouping variables are listed as their properties (*Properties*).

In addition to grouping, the dimensional resolution of information represents a significant structural pattern. Figure 3 (a., d., and e.) shows the increasing

resolution of the initially non-dimensional information: *the AlphaArc company has a total of 73 employees*. This information does not change subsequently, but the single value is broken down into individual values per store and then per store and department. Each dimension along which the information is resolved is listed as part of the variable (*Dimensions*). The DataDesc schema allows individual specification of the meaning (*Name*, *Description*, *URI*), index type (*Data Type*), and index range (*Has Minimum Value*, *Has Maximum Value*, *Value Set*, and *Value Increment*) of the dimensions. In this context, the combination of dimension indexes is unique, which is why it acts as a key and enables the unique identification and retrieval of each individual value. At the same time, an individual context is defined for each value: *15 employees is the team size of the sales department in the London store*, for example.

The structural description provides not only information about the context of values but also about data access mechanisms that might be expected by interface functions (see Figure 3 (c.)). For the structure of the *company list*, which can be, e.g., in the form of a *Python dictionary*, *pandas DataFrame*, *Java HashMap*, or *SQL table*, the variable *Name* was determined as a key index due to the identifying character of its values. As a dimension of the *company list*, the variable *Name* allows access to individual datasets.

Figure 3 (f.) shows another structure that combines grouping and resolution by adding the third dimension *year* to the resolution of the *number of employees* based on the two dimensions of *store* and *department*. Here, the total number of 73 employees is not broken down further, but put in the context of a specific year, e.g., *15 employees is the team size of the sales department in the London store in the year 2010*. Together with uniform information for, e.g., the years 2015 and 2020, this third dimension turns the dataset into a time series.

3.2 DataDesc Exchange Format and Utilities

In addition to the schema for describing interfaces, the exchange format for the integrated storage and flexible subsequent use of software metadata represents the second core component of the DataDesc ecosystem. The OpenAPI specification was chosen as its foundation, as it allows a programming language-agnostic description of software that is usable by both humans and computers. A software is described in a single OpenAPI-conforming YAML file. Its basic structure consists of a hierarchical object tree, that is subdivided into the two sections of *info* and *components* (compare Figure 2). In the *info* section, all general information, for example along a general software metadata schema such as CodeMeta, can be accommodated. If metadata elements are required for this that are not provided for in the OpenAPI specification, they can be added by means of *x-attribute* extensions without violating the standard. In the *components* section, the technical interface metadata, as described by the DataDesc schema, for example, can be specified. The *x-attributes* again provide the opportunity to compensate for missing OpenAPI metadata elements. In addition, they form the basis for using the standard not only to describe REST-compliant interfaces; they can also be used to arrange and annotate code elements such as classes,

functions, and parameters in the hierarchical object tree according to individual software interface designs.

In order to support the description of software based on its general properties and the transfer of this information into the exchange format chosen in the DataDesc approach, a *browser-based input form* was added to the ecosystem. The metadata fields of the form thereby map to the CodeMeta standard, as this is already widely used and can be applied across research domains. Thus, the uncomplicated input tool is an alternative to the JSON-based CodeMeta generator [64] and fits seamlessly into the DataDesc approach.

Unlike the general metadata, the technical documentation is produced directly in the source code of a program, which is why the definition of a machine-actionable formatting of this information, as well as its automated parsing and transfer into the exchange format must be made individually for each programming language. In the context of Python software, code components related to the interface are supplemented by decorators and individually marked up by means of DataDesc schema elements. A *Python parser* specifically developed for this schema extracts both the relevant code structures and their metadata and automatically generates the hierarchical object tree from them, which is then stored in an exchange format-compliant file.

The DataDesc utilities are complemented by a *tool for merging* the DataDesc files, so that the entire description of a software can be represented in a single concise documentation file that is easily exchangeable. Its OpenAPI conformity also ensures high interoperability, as a multitude of publicly-available tools can be applied to it [42, 65].

3.3 DataDesc Publication Pipelines

Making it possible for developers to create software metadata and documentation only once and then flexibly reuse it is one of the main objectives of the DataDesc approach. Against this background, technical processes are defined and, where necessary, supported with scripts that enable the collected information to be disseminated on software publication platforms. These publication pipelines are unique to each platform and subject to automation. To upload data to any of the publication platforms mentioned below, a free user account must first be created.

The OpenAPI-conforming YAML file can be uploaded and published on SwaggerHub [52] via its GUI or API, without any need for modifications. To publish the description on GitHub [43], it is sufficient to add the file to the software's versioned repository and reference it in the central README file. To make the documentation more visually-appealing, a link to a SwaggerHub-hosted documentation page can be included. The registration of Python-based software and its metadata in the Python Package Index (PyPi) [45] has been fully automated. With a DataDesc script utilizing restructuring and conversion tools, the YAML file and corresponding software source code are reformatted, uploaded, and published on the platform. The publication on ReadTheDocs [46] can also ingest information based on a DataDesc exchange file. In order to upload

the documentation in the appropriate format, it must first be created using, for example, Sphinx with its extension for the parsing of OpenAPI specifications. Then, a GitHub repository comprising the generated documentation can be imported. The ORKG [56, 57, 58] provides a GUI and an API for uploading software metadata, which can be entered manually into a form. In addition, a script was added to the DataDesc ecosystem to automate the translation of the exchange format into the ORKG template structures [66]. Currently, this mapping must still be performed individually by each user. However, work is underway to include this functionality in the ORKG. As part of the development of the Open Energy Research Graph [67], efforts are being made to ensure that the exchange format can also be processed directly within the OEP [32].

4 Application Case

In this section, the application of the DataDesc approach to a research software is discussed. For this, the open source FINE framework [12] from the *Energy Transformation Pathway Optimization Suite* (ETHOS) was chosen to illustrate a use case that is both realistic in terms of complexity and intriguing with respect to the interfaces provided. Using selected excerpts from the created interface documentation shown in Figure 4, the OpenAPI-compliant syntax of the YAML file generated using the DataDesc utilities is presented and the semantic expression capabilities of the DataDesc schema are assessed. To allow for a more in-depth review of the entire DataDesc approach, all code and documentation files created as part of this example application are published in the DataDesc repository [59].

FINE is a Python package with a five-year development history originating in the research domain of energy systems analysis [12]. It enables the modeling, optimization, and evaluation of energy systems. In addition to accounting for technical and environmental constraints, its optimization also seeks to minimize total annual energy system costs. It supports the creation and computation of spatially, temporally, and technologically highly-resolved models while integrating complexity-reduction techniques to shorten computation times. In its current version, 2.2.2 from 2022, the framework includes around 20,000 lines of code (excluding blank lines) and 10,000 lines of code documentation. Although the source code of the software project is hosted on GitHub [68], the user documentation is published on ReadTheDocs [69]. The documentation pages are based on the inline docstrings in the source code and were automatically generated using the Sphinx package. In addition, a short entry in the OEP's software framework list was written for the framework [70].

The FINE software is based on a central data model, namely the *Energy System Model* (ESM), which is represented by the ESM and component classes. It holds multidimensional information pertaining to the energy system under investigation and comprises all characteristics of its components, e.g., for the provision, transmission, and storage of energy. As input, besides basic parameters

for calculation control, the software requires the general conditions of the energy system and the techno-economic parameters of its components. As output, it provides information for the design and operation of a minimum-cost energy system. As the ESM incorporates all of this data, it simultaneously serves as the software's input and output data model.

As noted in the general schematic in Figure 1, the FINE interface offers the possibility of reading the input data from files or having them transferred by preceding software. However, in the second case the information can be gathered step by step in the data model classes; for file-based information transfer, a single complex file containing all parameters must be provided. Both Excel and NetCDF file formats are accepted for this purpose. On the output side of the interface, the result data can also be saved in Excel or NetCDF form or visually depicted using a range of plotting functions.

The FINE interfaces for reading Excel and NetCDF files are implemented by one function each, which mainly obtain a path to the respective input file. Here, DataDesc offers the possibility, in addition to the superficial description of the string variables, of going into depth and also describing the necessary internal structures of the input files (cf. Figure 4, lines 39-50). Thus, for the NetCDF interface, the control parameters and input variables were documented in the clearly structured hierarchical data format, which arranges the information according to entity types and in each case lists their attributes in accordance with their different dimensional characteristics. The documentation of the Excel data structure required more effort, as it does not group the data by entities, but distributes them to different spreadsheets depending on their dimensional resolutions. The resulting tables, in which the multi-dimensional attribute characteristics of different entities are mapped by means of different index columns, required precise documentation to define the boundaries between individual datasets. In both cases, documentation could be created to help users understand the given file structures and arrange their own input data accordingly. The documentation effort depended on the straightness and non-ambiguity of the data model structures.

For the documentation of the programmatic interface, the constructors of the ESM and the component classes were described using DataDesc. Here, the use of Python's dataclass decorators and minor code adjustments to the constructors enabled the transformation of previously free-text docstrings into unambiguous key-value pairs. Variable names, comments, types, roles, and default values could easily be mapped to the DataDesc annotation syntax (line 7 ff). For Python native types, the variable types were annotated directly into the code using type hints. For custom types, such as Panda's dataframes, these annotations, including more detailed structural information, were written in the decorators. In addition, the value range constraints that some variables are subject to could also be integrated into the documentation with the metadata elements contained in the DataDesc schema (lines 24-25). Furthermore, the permitted value sets and also complex data structures of the FINE variables could be described in detail (lines 12-18, 32-34). To minimize the documentation effort, value sets and

```yaml
openapi: 3.0.0
info: ### Info section
  title: FINE - A Framework for Integrated Energy System Assessment
  version: 2.2.2
  x-first-release: '2018-11-12'
  x-programming-lang: Python
components: ### Components section
  Component: ### FINE Component class
    description: The Component class includes...
    properties: ### Class Variables
      capacityMax:
        x-dimensions: &id001 ### Referencable inner Pandas Dataframe structure
          location:
            ItemMinimumValue: 0
            UnitType: spatial identifier
          time:
            ItemMinimumValue: 0
            UnitType: temporal identifier
  EnergySystemModel: ### FINE Energy System Model class
    properties: ### Class Variables
      numberOfTimeSteps:
        type: integer
        x-DefaultValue: 8760
        x-MinimumValue: 0 ### Allowed value range
        x-ExclusiveMinimum: true
    required: ### List of required parameters
    - numberOfTimeSteps
    x-functions: ### Class functions
      aggregateTemporally:
        properties: ### Function parameters
          clusterMethod:
            x-ValueSet: ### Allowed value set
            - averaging
            - k_means
            x-VariableRole: input ### Input parameter
      removeComponent:
        return: ### Return value
          $ref: '#/components/schemas/Component' ### Referencing data structure
      readNetCDFtoEnergySystemModel:
        properties: ### Function parameters
          filePath: ### Path to external file
            type: string
            x-FileFormat: NetCDF
            x-NetCDFFolders: ### Inner structure of referenced NetCDF file
              Input Data: ### Data folders
              - Conversion
              - Storage
              Parameters: ### Control parameters
              - numberOfTimeSteps
              - verboseLogLevel
```

Figure 4: Compilation of selected lines of code from the YAML file generated to document the FINE interfaces to represent the OpenAPI-compliant syntax and its semantics within the DataDesc schema.

data structures that apply to several variables were documented only once and then referenced repeatedly (lines 12 and 38).

Content dependencies, like the *costUnit* parameter of the ESM class, which determines the currency and units of all monetary variables, can also be expressed through referencing. The description of procedural dependencies, in which the value of a variable influences the software-internal calculation processes, must be represented so far as free text comments. An example for this is the component class that contains the Boolean parameter *hasCapacityVariable*, which, if set to *true*, causes the *capacityVariableDomain* and *capacityPerPlantUnit* variables to be ignored in the calculations. Work is currently underway to formally integrate this form of dependencies into the schema. Another dependency type results from the fact that FINE integrates the *tsam* [71, 72] library for the purpose of temporal data aggregation and partially maps the external interface within its own interface. In a function of the ESM class, for example, the parameter aggregation method can be selected (lines 31-34). The permitted value set, which includes options like *averaging* or *k-means*, is specified by the external library and manually included in the FINE documentation. In the future, the documentation of independent programs could be integrated and reused automatically if they are also made machine-actionable by means of the DataDesc schema.

5 Summary and Conclusions

The FAIR principles and their adaptations to research software have received much attention and support. To effectively reuse a software, the software itself and its interfaces must be clearly defined and made understandable, ideally in a machine-actionable manner. However, most research software today is not documented or published in a way that provides detailed and machine-actionable interface descriptions. Instead, software metadata is often focused on the compact provision of general information, whereas documentation pages, including detailed, technical information are primarily in natural language and not machine-actionable.

Therefore, the DataDesc ecosystem was presented in this article as an approach to describing the data models of software interfaces using detailed and machine-actionable metadata and as an extension to existing research software metadata. In pointing out that there must be a differentiation between data structures and data formats, it was shown how to consistently describe data structures, and by this support the interoperability of software to other programs and data files. In addition to a specialized metadata schema, an exchange format and support tools for the easy collection and automated publishing of software documentation were introduced. Using the FINE framework as an example, the practical applicability of DataDesc and its limitations were shown. It is hoped that DataDesc will facilitate the description of software interfaces with detailed and machine-actionable metadata enough to make it common practice, leading to increased interoperability, findability, and, therefore, reusability of research software.

In future research, DataDesc is to be enhanced and applied to the description of datasets. Furthermore, extending DataDesc to programming languages beyond Python would be an interesting research direction. With both software interfaces and data being described with DataDesc, the foundation for automatically composing and executing computational workflows has been laid, which will hopefully increase the reuse of research software and the reproducibility of computational research in the future.

6 Credit statement

Conceptualization: P.K., J.G., O.K., and S.F.; methodology: P.K., J.G., O.K., S.F., D.N., J.S., R.P., and F.E.; software: D.N., J.S., and O.K.; validation: R.P., P.K.; investigation: P.K., J.G., O.K., D.N., J.S., R.P., and S.F.; data curation: J.S.; writing – original draft: P.K., J.G., O.K., D.N., J.S., R.P. and S.F.; writing - review & editing: P.K., J.G., O.K., D.N., J.S., R.P., S.F., F.E., N.P., J.W., A.N., S.A., and D.S.; visualization: P.K., J.G.; supervision: D.S., S.A., and A.N.; project administration: P.K., O.K.; funding acquisition: D.S., S.A., and A.N.

7 Acknowledgements

The authors would like to thank the Federal Ministry for Economic Affairs and Energy of Germany (BMWi) for supporting this work with a grant for the project LOD-GEOSS (03EI1005B).

Furthermore, the authors are grateful to the German Federal Government, the German State Governments, and the Joint Science Conference (GWK) for their funding and support as part of the NFDI4Ing and the NFDI4Energy consortia. Funded by the German Research Foundation (DFG) – 442146713; 501865131.

In addition, the work was supported by the Lower Saxony Ministry of Science and Culture within the Lower Saxony "Vorab" of the Volkswagen Foundation under Grant 11-76251-13-3/19–ZN3488 (ZLE), and by the Center for Digital Innovation (ZDIN).

This work was also supported by the Helmholtz Association as part of the program "Energy System Design".

References

[1] Hartwig Anzt, Eileen Kuehn, and Goran Flegar. "Crediting pull requests to open source research software as an academic contribution". In: *Journal of Computational Science* 49 (2021), p. 101278.

[2] Arfon M Smith, Daniel S Katz, and Kyle E Niemeyer. "Software citation principles". In: *PeerJ Computer Science* 2 (2016), e86.

[3] Aidan Kelley and Daniel Garijo. "A framework for creating knowledge graphs of scientific software metadata". In: *Quantitative Science Studies* 2.4 (2021), pp. 1423–1446.

[4] *The CodeMeta Project*. URL: https://codemeta.github.io/ (visited on 01/16/2023).

[5] Stephan Druskat et al. "Software publications with rich metadata: state of the art, automated workflows and HERMES concept". In: *arXiv preprint arXiv:2201.09015* (2022).

[6] Anna-Lena Lamprecht et al. "Towards FAIR principles for research software". en. In: *Data Science* 3.1 (Jan. 2020), pp. 37–59. ISSN: 2451-8484. DOI: 10.3233/DS-190026. URL: https://content.iospress.com/articles/data-science/ds190026 (visited on 01/20/2021).

[7] Ted Habermann. "Metadata and reuse: Antidotes to information entropy". In: *Patterns* 1.1 (2020), p. 100004.

[8] Mark D Wilkinson et al. "The FAIR Guiding Principles for scientific data management and stewardship". In: *Scientific data* 3.1 (2016), pp. 1–9. ISSN: 2052-4463.

[9] Michelle Barker et al. "Introducing the FAIR Principles for research software". In: *Scientific Data* 9.1 (2022), pp. 1–6.

[10] Daniel S. Katz, Morane Gruenpeter, and Tom Honeyman. "Taking a fresh look at FAIR for research software". English. In: *Patterns* 2.3 (Mar. 2021). ISSN: 2666-3899. DOI: 10.1016/j.patter.2021.100222. URL: https://www.cell.com/patterns/abstract/S2666-3899(21)00036-2 (visited on 04/21/2021).

[11] Wilhelm Hasselbring et al. "From FAIR research data toward FAIR and open research software". en. In: *it - Information Technology* 62.1 (Feb. 2020), pp. 39–47. ISSN: 1611-2776, 2196-7032. DOI: 10.1515/itit-2019-0040. URL: https://www.degruyter.com/view/journals/itit/62/1/article-p39.xml (visited on 12/02/2020).

[12] Lara Welder et al. "Spatio-temporal optimization of a future energy system for power-to-hydrogen applications in Germany". In: *Energy* 158 (2018), pp. 1130–1149. ISSN: 0360-5442. DOI: https://doi.org/10.1016/j.energy.2018.05.059. URL: https://www.sciencedirect.com/science/article/pii/S036054421830879X.

[13] *Dublin Core Metadata Initiative (DCMI) Metadata Terms*. URL: https://www.dublincore.org/specifications/dublin-core/dcmi-terms/ (visited on 08/11/2022).

[14] *DataCite Metadata Schema*. URL: https://schema.datacite.org/ (visited on 08/11/2022).

[15] Marcia Lei Zeng and Jian Qin. *Metadata*. Third edition. London, Great Britain: Facet Publishing, 2022. ISBN: 978-1-78330-588-9.

[16] Stephan Druskat et al. *Citation File Format*. Version 1.2.0. Aug. 2021. DOI: 10.5281/zenodo.5171937. URL: https://doi.org/10.5281/zenodo.5171937.

[17] *The Software Description Ontology - Release May 3rd, 2021*. URL: https://w3id.org/okn/o/sd (visited on 01/16/2023).

[18] Yolanda Gil, Varun Ratnakar, and Daniel Garijo. "OntoSoft: Capturing Scientific Software Metadata". In: *Proceedings of the 8th International Conference on Knowledge Capture*. K-CAP 2015. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 1–4. ISBN: 978-1-4503-3849-3. DOI: 10.1145/2815833.2816955. (Visited on 06/22/2021).

[19] *OntoSoft Portal*. URL: https://www.ontosoft.org/portal/ (visited on 01/09/2023).

[20] Daniel Garijo et al. "OKG-Soft: An Open Knowledge Graph with Machine Readable Scientific Software Metadata". In: *2019 15th International Conference on eScience (eScience)*. Sept. 2019, pp. 349–358. DOI: 10.1109/eScience.2019.00046.

[21] *MINT Model Explorer*. URL: http://models.mint.isi.edu (visited on 01/16/2023).

[22] Jon Ison et al. "Tools and data services registry: a community effort to document bioinformatics resources". In: *Nucleic Acids Research* 44.D1 (Jan. 2016), pp. D38–D47. ISSN: 0305-1048. DOI: 10.1093/nar/gkv1116. URL: https://doi.org/10.1093/nar/gkv1116 (visited on 04/14/2021).

[23] Jon Ison et al. "The bio.tools registry of software tools and data resources for the life sciences". In: *Genome Biology* 20.1 (Aug. 2019), p. 164. ISSN: 1474-760X. DOI: 10.1186/s13059-019-1772-6. URL: https://doi.org/10.1186/s13059-019-1772-6 (visited on 02/05/2021).

[24] *bio.tools*. URL: http://bio.tools (visited on 01/16/2023).

[25] *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language - W3C Recommendation 26 June 2007*. URL: https://www.w3.org/TR/wsdl/ (visited on 01/18/2023).

[26] *Web Application Description Language - W3C Member Submission 31 August 2009*. URL: https://www.w3.org/Submission/wadl/ (visited on 01/18/2023).

[27] *OpenAPI Specification v3.1.0*. URL: https://spec.openapis.org/oas/v3.1.0 (visited on 01/18/2023).

[28] *OWL-S: Semantic Markup for Web Services*. URL: https://www.w3.org/Submission/OWL-S/ (visited on 01/16/2023).

[29] *Functional Mock-up Interface Specification, version 3.0, 2022-05-10*. URL: https://fmi-standard.org/docs/3.0/ (visited on 01/18/2023).

[30] Jan Schwarz and Sebastian Lehnhoff. "Ontological Integration of Semantics and Domain Knowledge in Energy Scenario Co-simulation". en. In: *Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*. Vienna, Austria: SCITEPRESS - Science and Technology Publications, 2019, pp. 127–136. ISBN: 978-989-758-382-7. DOI: 10.5220/0008069801270136.

[31] *openmod initiative's Wiki - Open Models*. URL: https://wiki.openmod-initiative.org/wiki/Open_Models (visited on 01/16/2023).

[32] *Open Energy Platform (OEP) - Model Factsheets*. URL: https://openenergy-platform.org/factsheets/models/ (visited on 01/16/2023).

[33] *PyPSA: Python for Power System Analysis*. URL: https://pypsa.readthedocs.io/en/latest/index.html (visited on 05/26/2023).

[34] Benjamin D. Lee. "Ten Simple Rules for Documenting Scientific Software". In: *PLOS Computational Biology* 14.12 (Dec. 20, 2018), e1006561. ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1006561. URL: https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1006561 (visited on 11/18/2022).

[35] *Javadoc Tool*. URL: https://www.oracle.com/java/technologies/javase/javadoc.html (visited on 01/23/2023).

[36] *Perldoc Browser 5.36.0*. URL: https://perldoc.perl.org/perlpod (visited on 01/23/2023).

[37] *Doxygen*. URL: https://www.doxygen.nl/ (visited on 01/23/2023).

[38] *sphinx.ext.napoleon – Support for NumPy and Google style docstrings*. URL: https://www.sphinx-doc.org/en/master/usage/extensions/napoleon.html (visited on 01/23/2023).

[39] *MkDocs - Project documentation with Markdown*. URL: https://www.mkdocs.org/ (visited on 01/23/2023).

[40] *Sphinx - Python Documentation Generator*. URL: https://www.sphinx-doc.org/en/master/ (visited on 01/23/2023).

[41] *roxygen2 7.2.3*. URL: https://roxygen2.r-lib.org/ (visited on 01/23/2023).

[42] *Swagger - API Documentation and Design Tools for Teams*. URL: https://swagger.io/ (visited on 01/23/2023).

[43] *GitHub - Let's build from here*. URL: https://github.com/ (visited on 01/24/2023).

[44] *Package repository for anaconda*. URL: https://anaconda.org/anaconda/repo (visited on 01/24/2023).

[45] *PyPi - Der Python Package Index*. URL: https://pypi.org/ (visited on 01/26/2023).

[46] *Read the Docs - Dokumentation erstellen, hosten und durchsuchen*. URL: https://readthedocs.org/ (visited on 01/24/2023).

[47] *GitLab - The DevSecOps Platform*. URL: https://gitlab.com/ (visited on 01/24/2023).

[48] *Bitbucket - Code and CI/CD, built for teams using Jira*. URL: https://bitbucket.org/ (visited on 01/24/2023).

[49] *Sourceforge - The Complete Open-Source and Business Software Platform*. URL: https://sourceforge.net/ (visited on 01/25/2023).

[50] *Maven Central Repository Search*. URL: https://search.maven.org/ (visited on 01/24/2023).

[51] *NPM - Node Package Manager*. URL: https://www.npmjs.com/ (visited on 01/25/2023).

[52] *SwaggerHub - Search public APIs and Domains in SwaggerHub*. URL: https://app.swaggerhub.com/search (visited on 01/30/2023).

[53] *GitBook - Where technical teams document*. URL: https://www.gitbook.com/ (visited on 01/24/2023).

[54] *The Comprehensive R Archive Network (CRAN)*. URL: https://cran.r-project.org/ (visited on 01/24/2023).

[55] *Github Pages - Websites for you and your projects*. URL: https://pages.github.com/ (visited on 01/25/2023).

[56] *Open Research Knowledge Graph (ORKG)*. URL: https://orkg.org/ (visited on 01/27/2023).

[57] Sören Auer et al. "Improving Access to Scientific Literature with Knowledge Graphs". In: *Bibliothek Forschung und Praxis* 44.3 (2020).

[58] Markus Stocker et al. "FAIR Scientific Information with the Open Research Knowledge Graph". In: *FAIR Connect* 1.1 (Jan. 11, 2023). DOI: 10.3233/FC-221513. (Visited on 01/11/2023).

[59] *DataDesc - An ecosystem for machine-actionable software metadata*. URL: https://github.com/FZJ-IEK3-VSA/DataDesc (visited on 01/31/2023).

[60] Sandra Heiler. "Semantic interoperability". In: *ACM Computing Surveys (CSUR)* 27.2 (1995), pp. 271–273.

[61] Meisam Booshehri et al. "Introducing the Open Energy Ontology: Enhancing data interpretation and interfacing in energy systems analysis". In: *Energy and AI* 5 (2021), p. 100074.

[62] *FINE's Energy System Model Class*. URL: https://vsa-fine.readthedocs.io/en/master/sourceCodeDocumentation/energySystemModelDoc.html (visited on 01/09/2023).

[63] *IAMC's Pyam Data Model*. URL: https://pyam-iamc.readthedocs.io/en/stable/data.html (visited on 01/09/2023).

[64] *CodeMeta generator*. URL: https://codemeta.github.io/codemeta-generator/ (visited on 01/27/2023).

[65] *OpenAPI.Tools*. URL: https://openapi.tools/ (visited on 01/27/2023).

[66] Oliver Karras et al. "Researcher or Crowd Member? Why not both! The Open Research Knowledge Graph for Applying and Communicating CrowdRE Research". In: *IEEE 29th International Requirements Engineering Conference Workshops*. IEEE. 2021.

[67] *Open Energy Knowledge Graph (OEKG)*. URL: https://github.com/OpenEnergyPlatform/oekg (visited on 01/30/2023).

[68] *FINE - Framework for Integrated Energy System Assessment*. URL: https://github.com/FZJ-IEK3-VSA/FINE (visited on 02/11/2023).

[69] *Welcome to FINE's documentation! FINE - A Framework for Integrated Energy System Assessment*. URL: https://vsa-fine.readthedocs.io/en/master/ (visited on 01/20/2023).

[70] *OEP Framework Factsheet - Framework Framework for Integrated Energy System Assessment (FINE)*. URL: https://openenergy-platform.org/factsheets/frameworks/164/ (visited on 02/18/2023).

[71] *tsam - Time Series Aggregation Module*. URL: https://github.com/FZJ-IEK3-VSA/tsam (visited on 02/21/2023).

[72] Maximilian Hoffmann et al. "Typical periods or typical time steps? A multi-model analysis to determine the optimal temporal aggregation for energy system models". In: *Applied Energy* 304 (2021), p. 117825. ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2021.117825. URL: https://www.sciencedirect.com/science/article/pii/S0306261921011545.