Semantic Programming Framework for developing ontology-controlled Applications

Lars Vogt[a, *], Roman Baum[a], Christian Köhler[a], Sandra Meid[a], Björn Quast[b] and Peter Grobe[b]
[a]*Institut für Evolutionsbiologie und Ökologie, Rheinische Friedrich-Wilhelms-Universität Bonn, An der Immenburg 1, 53121 Bonn, Germany*
E-mails: lars.m.vogt@googlemail.com, R.Baum@leibniz-zfmk.de, c.koehler@zfmk.de, s.meid.zfmk@uni-bonn.de
[b]*Biodiversity Informatics, Zoologisches Forschungsmuseum Alexander Koenig, Adenauerallee 160, 53113 Bonn, Germany*
E-mails: B.Quast@leibniz-zfmk.de, P.Grobe@leibniz-zfmk.de

Abstract. We demonstrate that ontologies are not restricted to modeling a specific domain, but can be used for programming as well. We introduce the Semantic Programming Ontology (SPrO) and its accompanying Java-based middleware, which we use as a semantic programming language. SPrO provides ontology instances as well as annotation, object, and data properties, which function as commands, attributes, and variables that the middleware interprets. We use SPrO for describing data-centric Semantic Web applications. Each description forms an ontology itself, i.e., the application's source code ontology (SCO). The Java-based middleware produces the respective application and controls its behavior based on the descriptions contained in the SCO. The middleware thus functions like an interpreter that interprets the descriptions in the SCO in reference to SPrO. It treats descriptions as specifications of the application and dynamically executes them. The goal of our semantic programming approach is to have a development framework that not only seamlessly integrates the RDF world with the HTML world but also allows domain experts to develop their own data-centric Semantic Web applications with as little programming experience required as possible. SPrO and its accompanying Java-based middleware and its interface are available from (https://github.com/SemanticProgramming).

Keywords: Semantic Programming Ontology (SPrO), semantic programming, semantic programming language, ontology-controlled application, Semantic Web application

1. Introduction

In the age of Big Data, Linked-Open-Data, and the Semantic Web, efficiently managing and organizing data has become key to **data exploration** and **eScience**, a newly emerged driving force for scientific progress in data-rich fields of empirical research [1]. eScience requires data not only to be maximally findable, accessible, interoperable and reusable (FAIR guiding principle [2]), but also computer-parsable [3–5]. As a consequence, new applications and services have been developed, many of which utilize Semantic Web technologies and web-based data-centric applications, such as web content management systems. Ontologies and other controlled vocabularies have taken a central role in this context because they provide the required standardized semantic structure for data and metadata to become comparable and computer-parsable (e.g., [4,6–8]).

Ontologies are dictionaries that can be used for describing a certain reality. They consist of labeled classes with commonly accepted definitions that are formulated in a highly formalized canonical syntax and standardized format, such as the Web Ontology Language (OWL) serialized to the Resource Description Framework (RDF), with the goal to yield a lexical or taxonomic framework for knowledge representation [9]. Each ontology class possesses its own Uniform Resource Identifier (URI), through which it can be identified and individually referenced.

Ontologies can be documented and represented in form of class-based semantic graphs. A semantic

[*]Corresponding author. E-mail: lars.m.vogt@googlemail.com.

graph is a network of RDF/OWL-based triple statements, in which a given URI takes the *Object* position in one triple statement (i.e., a statement consisting of *Subject*, *Predicate,* and *Object*) and the *Subject* position in another triple statement and thus connecting these statements to form a graph. Because information about particular entities can be represented as a semantic graph as well, we distinguish class-based and instance-based semantic graphs.

Ontologies contain commonly accepted domain knowledge about specific kinds of entities together with their properties and relations in form of classes defined through universal statements [10,11]. Ontologies in this sense do not include statements about particular entities. Statements about particular entities are assertional statements[1]. If assertional statements are grounded in empirical knowledge that is based on observation and experimentation, we refer to them as empirical data. In an assertional statement, a particular entity can be referred to by providing it its own URI, and its class affiliation can be specified by referencing the URI of the respective class. Empirical data can thus be formulated in OWL and documented in form of instance-based semantic graphs (representing data as an instance-based instead of a class-based semantic graph has many advantages [12]). As a consequence, not every OWL file and not every semantic graph is an ontology—it is an ontology if and only if it limits itself to expressing only universal statements about kinds of entities [11].

A knowledge base, in contrast, consists of a set of ontology classes that are populated with particular entities and assertional statements about these entities (i.e., data) [11]. Ontologies, therefore, do not represent knowledge bases, but are part of them and provide a means to structure them [13]. In other words, a knowledge base links data statements in form of instance-based semantic graphs to specific ontology classes, with the result that the data statements become semantically transparent because they reference the ontology classes that each of its described particular entities instantiates. Referencing the ontology class of each described particular entity also substantially increases the computer-parsability of the data statements and the possibility to reason over them, thereby taking full advantage of the power of Semantic Web technologies. Ontologies thus provide a framework for establishing standards that improve the integration and interoperability of data and metadata statements, all of which is much needed in eScience [3,5,14].

Unfortunately, not many web content management systems have implemented ontologies and semantic graphs to their full potential by using them in a knowledge base to document data statements. The overwhelming majority of applications of ontologies in the life sciences has been restricted to semantically enriching documents and annotating database contents by using the URIs of ontology classes as values within tables of relational databases, instead of documenting and communicating data as instance-based semantic graphs. This is not due to technological limitations and restrictions. Tuple stores that store triple statements based on RDF's syntax of *Subject*, *Predicate* and *Object* are capable of handling large volumes of triples. These triples may express data and metadata statements as well as underlying data schemes in form of semantic graphs. Semantic technology facilitates detailed information retrieval of information represented as either class-based or instance-based semantic graphs through SPARQL endpoints [15] and inferencing over semantic graphs through reasoners [16].

Despite these obvious advantages of tuple stores and RDF/OWL-based data solutions, they nevertheless have yet to replace conventional databases such as MySQL or PostgreSQL in rank as the prime database technology for developing Semantic Web applications for scientific data. In search of an explanation for this discrepancy, we believe that a lack of application development frameworks that are well integrated with RDF/OWL is responsible for this situation (for initial attempts, though not specifically developed for web content management systems, see, e.g., [17–25]). RDF and OWL, coupled with a tuple store, provide an efficient means to store and query data. Semantic instance-based graphs representing data and metadata statements can be readily consumed by various applications through a corresponding SPARQL endpoint. However, semantic graphs often posses a rather complicated structure. They are usually not as intuitively comprehensible for a human reader as data and metadata represented in conventional tables or entry forms. Thus it is not surprising that human readers generally are not interested in interacting with actual semantic graphs. As a consequence, semantic data-centric applications would have to hide the graphs from their users and, instead, provide more user-friendly representations of their data. Unfortunately, SPARQL endpoints only allow

[1] Description Logic (DL) distinguishes TBox and ABox expressions. TBoxes contain assertions on classes, whereas ABoxes contain assertions on instances. Class axioms expressed in OWL are TBox expressions. An ontology contains TBox expressions, whereas a knowledge base (see below) expressed in DL is constituted by TBox and ABox expressions [36].

interacting directly with a semantic graph and do not provide a user-friendly presentation of the data, as for instance through an HTML-based interface. What is required in order to increase the applicability of semantic graphs is a means for users to indirectly interact with them through data entry forms, tables and other ways of visualizing and interacting with data in intuitive ways.

Here, we introduce SPrO, the Semantic Programming Ontology, and its accompanying Java-based middleware. With them, we want to contribute to the development of a framework that will close the gap between computer-parsable data represented in form of semantic graphs on the one hand and a user-friendly visualization of data in form of HTML-based data entry forms on the other hand. SPrO enables users to describe the data-centric Semantic Web application they need for efficiently managing and disseminating data in an eScience-compliant way. The description of the respective application is formulated in OWL and stored as source code ontologies. The accompanying middleware functions as an interpreter that dynamically executes the description contained in the source code ontologies by interpreting it as a declarative specification. The overall goal of this semantic programming approach is to provide a one-stop framework for developing customized data-centric Semantic Web applications.

2. Semantic Programming

2.1. General Idea for Semantic Programming using SPrO

Within academia, the practical application of ontologies is usually restricted to providing URIs for annotating data and metadata statements or documenting them in form of semantic graphs for a specific scientific domain. With the Semantic Programming Ontology (SPrO) we extend this application and use ontologies in software programming. We apply SPrO like a programming language to specify and control data-centric Semantic Web applications. This is achieved by describing the application within a corresponding source code ontology (SCO) using terms from SPrO. By being able to integrate the description of the application's data model with the description of its graphical user interface (GUI) and the application's overall functionality, we realize another goal of ours: being able to implement changes to a data-centric application such as a web content management system without having to do programming in three different layers (i.e., database, middleware, and frontend) using three different sets of technologies. Using semantic programming, we only have to make changes to the corresponding SCO using terms from SPrO.

SPrO defines ontology resources in form of classes, individuals and properties that the accompanying Java-based middleware interprets as a set of commands, subcommands, and variables. The commands and subcommands are defined as annotation properties. Specific values and variable-carrying resources are defined as ontology individuals (i.e., instances of ontology classes). Additional object properties are used to specify relations between resources, and data properties are used for specifying numerical values or literals for resources that describe the application.

SPrO can be used to describe all features, workflows, database processes and functionalities of a data-centric application, including its GUI. These descriptions are formulated in form of annotations of ontology classes and ontology individuals and documented in the SCO of the application. Each annotation consists of a command followed by a value, index or resource and can be extended by axiom annotations that contain subcommands, values, and variables taken from SPrO. In case the descriptions are linked to ontology individuals of SCO, the annotations can also be extended by property assertions.

The middleware associated with SPrO reads the source code contained in the application's SCO and dynamically executes it in reference to SPrO. In other words, the commands and variables from SPrO are used for creating declarative specifications of the application, which the middleware interprets and dynamically processes on the fly—the specification thus runs directly and no intermediate programming step in another layer is required. We call this approach to programming **semantic programming**. Semantic programming involves the following elements (Fig. 1):

1) the **Semantic Programming Ontology** (**SPrO**), which we use like an ontology-based programming language;
2) an **application source code ontology** (**SCO**) that contains the description of all database processes, data views and data entry forms with input controls of the data-centric Semantic Web application we want to develop, including the data scheme underlying the application and the overall appearance and organization of its GUI;

3) the **Java-based middleware** associated with SPrO that functions as an interpreter, interpreting the SCO in reference to SPrO and providing information for the **frontend**, for which HTML5/CSS3 should be used, with a GUI that is based on the specifications in SCO, as well as a SPARQL endpoint;
4) a **tuple store framework** for storing not only SCO and SPrO but also all data and metadata statements produced by the users of the application. These statements are stored in form of semantic graphs. We use the **Jena tuple store**, which can be organized into several independent physical RDF stores, with each such store representing a separate **workspace**.

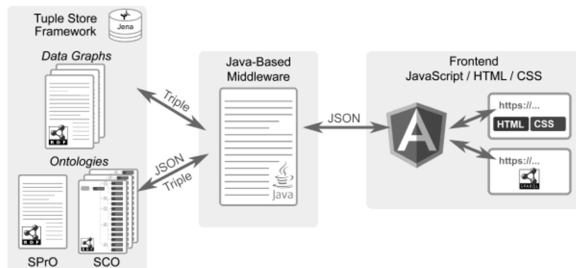

Fig. 1. Overall workflow of a data-centric Semantic Web application based on semantic programming. **Left:** Jena tuple store framework containing the data of the application as well as the Semantic Programming Ontology (SPrO) and the source code ontology (SCO) of the application. The data are stored in form of instance-based semantic graphs. SPrO provides the commands, subcommands, and variables used for describing the application. SCO contains the descriptions formulated using terms from SPrO. **Middle:** The Java-based middleware reads the descriptions contained in SCO and interprets them as the specification of the application. **Right:** The frontend, based on the JavaScript framework AngularJS, with HTML and CSS output for browser requests and access to a SPARQL endpoint for service requests.

The application with its tuple store framework forms a **knowledge base**. In addition to its web portal, all of its data and metadata statements can alternatively be accessed through the SPARQL endpoint of the application. Moreover, semantic reasoners can make inferences over the statements contained in the tuple store of the application.

Contrary to other development frameworks that utilize ontologies [17–25], SPrO and its accompanying Java-based middleware can be used to describe a particular data-centric Semantic Web application within a source code ontology that is specifically customized for the application. All information is contained in the application's tuple store framework, including SPrO, the application's SCO and all of its data and metadata statements. **The application and all of its data are thus fully self-describing.**

2.2. Example Descriptions from a Source Code Ontology

In the following, we give examples for how commands of SPrO and their subcommands are used within the application's SCO for describing functions and execution procedures of a data-centric Semantic Web application.

2.2.1. Sequences of Execution Steps

In our approach to semantic programming, many functions and processes of an application require the description of a specific sequence of commands and accompanying subcommands within the application's SCO. This can be accomplished by using a command annotation property from SPrO that triggers a certain type of execution step. Within the application's SCO, an index is assigned as a value to this annotation property, specifying the position of the corresponding execution step within the particular sequence of execution steps describing a particular process or feature for the application (Fig. 2). This overall scheme of annotating ontology individuals as well as ontology classes of the application's SCO is used for describing the various features, processes, and functions of the application.

2.2.2. Generating an Individual Resource of a Given Ontology Class

In various occasions, for instance, when creating a new data entry (see Fig. 2), individual instances must be generated for ontology classes that are defined in the application's SCO or in one of the external domain reference ontologies that the data-centric application references in its underlying data scheme. For example, when creating a new data entry, a set of resources must be generated that are mandatory for this type of data entry, including all named graph[2] instances required for organizing and managing all the triples associated with this entry.

[2] A named graph identifies a set of triple statements by adding the URI of the named graph to each triple belonging to this named graph, thus turning the triple into a quad. The Jena tuple store framework can handle such quadruples. The use of named graphs enables partitioning data in an RDF store.

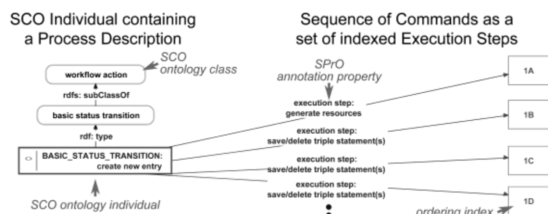

Fig. 2. Example for the annotation of a Source Code Ontology (SCO) individual that describes the basic status transition 'create new entry', a transition that is part of the life-cycle workflow of a data entry in a web-based semantic content management system. **Left:** The individual ontology resource from the application's SCO that contains the description of the status transition. **Right:** The set of commands that describe the status transition as a set of indexed execution steps using annotation properties from the Semantic Programming Ontology (SPrO). The white boxes to the right are the indices that specify the sequential order in which the execution steps must be processed (from low to high numbers, from A to Z). The first execution step '1A' triggers the generation of new resources, each of which receives its own URI, the following execution steps '1B' to '1D' save or delete specific triple statements. The commands together with their ordering indices are, in this case, annotations of an ontology individual of SCO, but can be annotations of an ontology class of SCO in another case. (The subcommands accompanying each execution step are not shown in this figure)

Using the SPrO annotation property *'execution step: generate resources'* in a description within the application's SCO specifies that this is an execution step command that triggers the generation of new resources. The index value assigned to the command specifies the position within the sequence of execution steps. The linked subcommands specify the ontology classes for which new instances must be generated when executing the step (Fig. 3). Each newly generated resource defines a corresponding SPrO variable-carrying resource that can be used for referencing the resource's URI in a later execution step. The instance of *'specimen'*, for example, a class which is referenced in Figure 3, is generated based on the SPrO annotation property *'generated resource of class [input_5]'* and can be referred to in subsequent execution steps through the SPrO variable-carrying resource '*SPrO_VARIABLE: generated resource [input_5]'*.

The SPrO annotation property *'generates resources for entry ID'* is used as a subcommand that defines for which entry ID the resources must be generated. This affects the URI of all resources generated during this execution step and references the entry's URI within the URI of the generated resource[3].

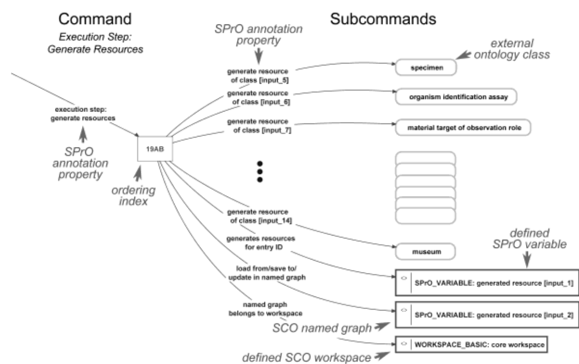

Fig. 3. Example of a Source Code Ontology (SCO) description of an execution step for generating instances of defined ontology classes. **Left:** The command, in form of a Semantic Programming Ontology (SPrO) annotation property, specifies that this execution step triggers the generation of new ontology resources. The value assigned to the command determines that it takes the 19AB position within the sequence of execution steps described in the application's SCO. **Right:** The set of subcommands that specify the ontology classes for which new instances must be generated, the entry ID of the data entry for which the resources are generated and the named graph and workspace where a set of triple statements must be stored indicating the class affiliation and the instance-status of each newly generated resource. Note that in this example the entry ID, as well as the named graph, is referred to by using the variable-carrying resources '*SPrO_VARIABLE: generated resource [input_1]'* and '*SPrO_VARIABLE: generated resource [input_2]'*, respectively, each of which has been defined in a previous resource generating execution step (e.g., execution step 1A in Fig. 2).

The SPrO annotation property *'load from/save to/update in named graph'* is used as subcommand that determines a particular named graph. The middleware automatically generates a set of triple statements that specify the class affiliation and the instance-status for each newly generated resource and stores it to this named graph. Finally, the SPrO annotation property *'named graph belongs to'* is used as subcommand that determines the workspace where this named graph is located.

2.2.3. Saving or Deleting a Specific Triple Statement

Saving and deleting specific triple statements is an essential process of any data-centric Semantic Web application that uses a triple store. This process is triggered using the SPrO annotation property *'execution save/delete triple statement(s)'* in a description within the application's SCO as an execution step command that triggers saving or deleting specific triple statements. Additional SPrO annotation properties are used as subcommands that specify a triple statement and the location where the triple statement must be stored to or deleted from in terms of named

[3] The URI of the generated resource is the combination of the entry's URI and the URI of the ontology class that the resource instantiates. This way, the resource's URI itself already provides information of its class affiliation and for which data entry it has been generated.

graph and workspace. The SPrO annotation property *'delete triple statement [BOOLEAN]'* can be used with the Boolean value *'true'* to indicate that the triple statement must be deleted (Fig. 4). If the triple statement must be saved, the respective Boolean subcommand is not used in the description of this execution step.

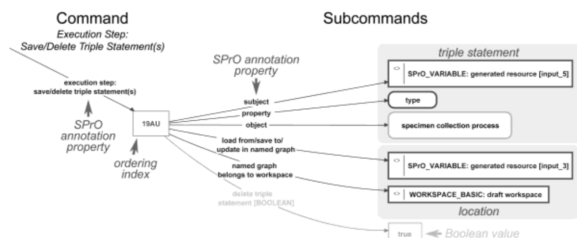

Fig. 4. Example of a Source Code Ontology (SCO) description of an execution step for saving/deleting a particular triple statement. **Left:** The command, in form of a Semantic Programming Ontology (SPrO) annotation property, specifies that this execution step triggers the saving (or deleting) of a particular triple statement. The value assigned to the command determines that it takes the 19AU position within the sequence of execution steps described in the application's SCO. **Right:** A set of subcommands specifies the particular triple statement to be saved/deleted in form of the SPrO annotation properties '*subject'*, '*property'* and '*object'*. Note that the resource assigned to '*subject'* is a SPrO variable-carrying resource that has been defined in a previous execution step. The triple that must be saved/deleted states that the newly generated resource instantiates the ontology class *'specimen collection process'*. Additional subcommands are used to specify the location to which the triple must be saved in terms of named graph and workspace. If the specified triple statement should be deleted instead of being saved, the SPrO annotation property *'delete triple statement [BOOLEAN]'* must be used and the value '*true'* be assigned to it (see transparent subcommand).

2.2.4. Copying and Deleting Named Graphs

Copying named graphs is a command that is triggered using the SPrO annotation property *'execution step: copy named graphs'* in a description within the application's SCO. Additional SPrO annotation properties are used as subcommands to specify the named graph that must be copied by either referencing the class to which the named graph belongs (*'load from/save to/update in named graph (this entry's specific individual of)'*) or the particular named graph resource (*'load from/save to/update in named graph'*). The SPrO annotation property *'copy from workspace'* is used as a subcommand for specifying the workspace where the named graphs are located that must be copied.

Since SPrO also allows the definition of variable-carrying named graph resources, which at their turn can contain a list of several resources and thus also several named graph resources, a variably-carrying resource can reference to a list of named graphs that must be copied by using the SPrO annotation property *'load from/save to/update in named graphs of this SPrO variable list'* (Fig. 5). The SPrO annotation property *'named graph belongs to workspace'* is used to specify the workspace to which the copied named graphs must be saved.

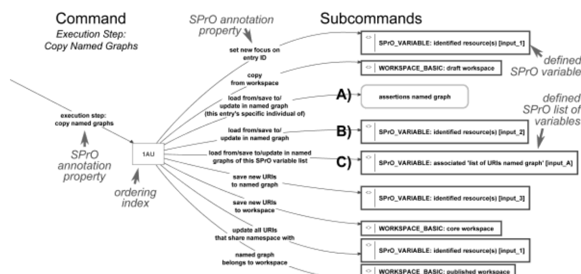

Fig. 5. Example of a Source Code Ontology (SCO) description of an execution step for copying named graphs. **Left:** The command, in form of a Semantic Programming Ontology (SPrO) annotation property, specifies that this execution step triggers the copying of named graphs. Its value determines that it takes the 1AU position within the sequence of execution steps. **Right:** A set of subcommands specifies the particular named graphs to be copied and the workspace in which they reside. The named graphs to be copied can be specified in reference to either **A)** a respective class of named graphs, in which case the middleware identifies the particular named graph resource (only applicable, if the data entry possesses exactly one named graph resource of this class), to **B)** a particular named graph resource (through a SPrO variable-carrying resource that references it, as for instance '*SPrO_VARIABLE: identified resource(s) [input_2]'*, which at its turn must have been defined in a previous execution step, for instance, as a result of a search) or to **C)** a list of particular named graphs (through a SPrO variable-carrying resource that is itself a named graph in which several named graph resources are listed, for instance, '*SPrO_VARIABLE: associated 'list of URIs named graph' [input_A]'*). Note that the subcommand at the top defines the focus to be on a specific entry ID. This information is used by the middleware to identify all entry specific individual resources for which only the class affiliation is known (e.g., *'load from/save to/update in named graph (this entry's specific individual of)'*).

If desired, not only the copied named graphs themselves but also all resources copied with them can receive new URIs. This is accomplished through referencing additional SPrO annotation properties in the description within the application's SCO. These properties are used for defining which URIs should be updated by indicating their namespaces (*'update all URIs that share namespace with'*).

Deleting entire named graphs, as opposed to deleting single triple statements in a named graph, is triggered through the SPrO annotation property *'execution step: delete named graphs'*, with the accompanying subcommands specifying the named graph to be deleted and the workspace they reside (Fig. 6). By setting the focus to a variable-carrying resource that is a named graph which contains the entry IDs of

several data entries (e.g., all draft versions of a given entry), a single execution step can delete all named graphs of a certain type of entry at once.

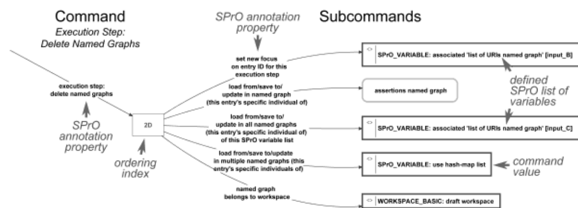

Fig. 6. Example of a Source Code Ontology (SCO) description of an execution step for deleting named graphs. **Left:** The command, in form of a Semantic Programming Ontology (SPrO) annotation property, specifies that this execution step triggers the deletion of named graphs. Its value determines that it takes the 2D position within the sequence of execution steps. **Right:** A set of subcommands specifies the particular named graphs to be deleted and the workspace in which they reside. The named graphs to be deleted can be specified in a similar manner as when copying them (see Fig. 5). However, when deleting several named graphs of which only the class affiliations are known, which in turn are listed in a SPrO variable-carrying resource that is a named graph, an additional subcommand must be added with the value '*SPrO_Variable: use hash-map list'* to indicate this.

2.2.5. Searching a Specific Resource Based on a Known Triple Statement

Searching specific resources based on a (partly) known triple statement is a command that is triggered using the SPrO annotation property *'execution step: search triple store'*, with the accompanying subcommands specifying the (partly) known triple and its location in terms of named graph and workspace. Using the SPrO annotation property *'search target'* as a subcommand, one can specify the position of the searched resource within the triple statement (Fig. 7). If we know nothing about the class affiliation of a resource within the triple, we use the '*SPrO_VARIABLE: ?'* value to indicate that this position in the triple statement must be left blank. The result of the search will be associated with a SPrO variable-carrying resource that is specified through the subcommand *'search target defines SPrO variable'*. If multiple hits are expected, the SPrO annotation property *'search target saved to 'list of URIs named graph' SPrO variable'* must be used together with *'multiple-hits-search [BOOLEAN]'* with the Boolean value *'true'*, which will save the list of found resources to a SPrO variable-carrying resource that is a named graph.

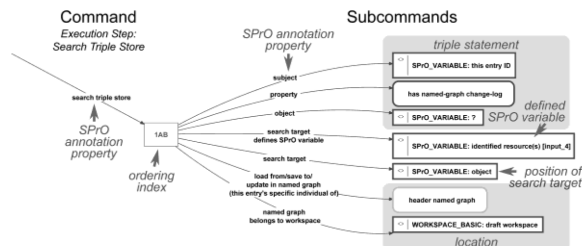

Fig. 7. Example of a Source Code Ontology (SCO) description of an execution step for searching a particular resource in the application's triple store framework. **Left:** The command, in form of a Semantic Programming Ontology (SPrO) annotation property, specifies that this execution step triggers the search for a specific resource within the application's triple store framework. Its value determines that it takes the 1AB position within the sequence of execution steps. **Right:** A set of subcommands specifies the (partly) known triple statement, its location in the triple store framework in terms of named graph and workspace, and the position that the searched resource takes within this statement. If we do not know the class affiliation of one of the resources from the triple, we can use the '*SPrO_VARIABLE: ?'* value to leave that position blank. The SPrO annotation property *'search target'* can be used as a subcommand to specify the position of the searched resource within the triple statement and the SPrO annotation property *'search target defines SPrO variable'* to specify the SPrO variable-carrying resource that can be used in subsequent execution steps to refer to the resource found during this execution step.

As a side note: variable-carrying resources can be defined not only during the generation of resources (see 2.6.2) or through searches, but also directly using the SPrO annotation property *'execution step: define variables'* as a command. Corresponding subcommands also allow adding or deleting particular resources to or from an already defined SPrO variable-carrying resource that tracks a list of resources.

2.2.6. If-Then-Else Conditions

If-then-else conditions can be specified through the command that uses the SPrO annotation property *'execution step: if-then-else statement'* in a description within the application's SCO. The SPrO annotation property *'has IF input value'* can be used as an accompanying subcommand that specifies a particular input value and the SPrO annotation property *'has IF target value'* for specifying a particular target value (both either a particular resource or a value/label). The SPrO annotation property *'has IF operation'* is used as a subcommand for specifying a particular defined SPrO *if*-operation (Fig. 8). Various SPrO *if*-operations are defined (see Table 1) which specify the criteria for which the if-then-else condition would be *'true'*. The SPrO annotation properties *'then:'* and

'else:'[4] are used for indicating to which execution step the application should proceed in case the condition is '*true'* or '*false'*, respectively. With the SPrO annotation property *'has THEN operation'* a particular defined SPrO operation can be triggered in case the condition is '*true'*.

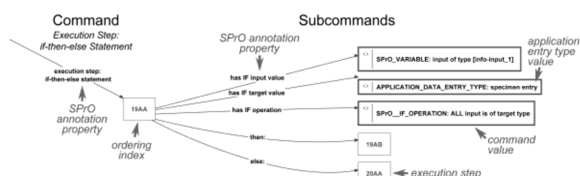

Fig. 8. Example of a Source Code Ontology (SCO) description of an execution step for specifying an if-then-else condition. **Left:** The command, in form of a Semantic Programming Ontology (SPrO) annotation property, specifies that this execution step triggers the specification of an if-then-else condition. Its value determines that it takes the 19AA position within the sequence of execution steps. **Right:** The SPrO annotation property *'has IF input value'* is used as a subcommand that specifies the *if*-input-value to be the value that is assigned to a specific SPrO variable-carrying resource that has been defined in a previous execution step. The SPrO annotation property *'has IF target value'* on the other hand is used for specifying that the target value is a defined value that indicates the application's data entry type *specimen entry*. The SPrO annotation property *'has IF operation'* is used as a subcommand with the value *'SPrO__IF_OPERATION: ALL input is of target type'*, which triggers the application to compare the input with the target value and returns '*true'* if they are identical and '*false'* if they differ. If '*true'*, the application will proceed with execution step 19AB, if '*false'* with 20AA.

Each if-then-else execution step only has a single *'else'* clause. If you want to execute a more complex if-then-else condition, you must concatenate several if-then-else execution steps.

2.3. Overall Expressivity of SPrO

At its current state of development, SPrO and its accompanying middleware can be used for describing a data-centric Semantic Web application such as a semantic web content management system in an SCO that is customized for the application. The description specifies workflows and database processes (i.e., storing, retrieving, searching and updating triples in the tuple store framework) as well as different data views with their corresponding HTML data entry forms and pages. SPrO and the middleware support basic functionalities like:

- input fields with auto-completion for ontology terms,
- input control with message-feedback,
- search and filtering of triples and of individual resources in the tuple store framework,
- semantic annotation of free texts,
- automatic provenance tracking and tracking of user input in a history-log,
- user administration with signup and login processes as well as session management,
- a publication life-cycle (draft > publish > revise > publish) of data entries and
- a SPARQL endpoint.

As the examples above indicate, the more complex specifications require the description of sequences of ordered execution steps and cannot be handled in a single command. Table 1 lists the label of all respective annotation properties from SPrO that serve as commands together with their description and, where applicable, also the corresponding Java method from the accompanying middleware (Table 1, rows with grey background). Each execution step annotation property usually possesses a set of associated annotation properties that serve as their subcommands, which are used in axiom annotations or property annotations to further specify the command in descriptions within the application's SCO. These are also listed in Table 1, directly below their respective command (Table 1, rows with white background). In some cases, SPrO value-carrying ontology instances that relate to the respective execution step are listed as well.

The underlying **Jena tuple store framework** is organized into different workspaces. Each workspace is an independent physical RDF store that can be accessed through the application's SPARQL endpoint. The different workspaces strictly separate data from administrative and application-centered information and can also be used for separating published data from draft data. This physical separation of different types of information increases overall data safety.

Each workspace can be further structured into various different **named graphs**, with each named graph having its own URI and instantiating a specific class of named graphs. This can be used to **differentially store triples** and hence structure a workspace into various instances of named graphs of different named graph classes, which at its turn not only facilitates data retrieval and increases data safety, but also allows **flexible and meaningful fragmentation of data** (for a discussion see [12,26]). Moreover, since a named graph identifies an entire semantic graph through its URI, named graphs can be used for **reification**, i.e., making statements about statements,

[4] The SPrO annotation property *'else'* must be used only if the application should NOT proceed with the next execution step in case the *if* condition is *'false'*.

such as when adding metadata to a single triple statement or an entire semantic graph. Furthermore, a named graph can also be used as a SPrO variable containing an ordered list of resources, which can be used for tracking specific resources during various application processes.

SPrO annotation properties for commands and their subcommands and relevant SPrO variables	Description	Java Method
execution step trigger	**This annotation property is used for specifying what triggers a sequence of consecutive execution steps, including specific requirements that must be met for triggering the sequence and specifying the starting execution step.**	
autocomplete for ontology	Specifies the ontology that provides the basis for the autocomplete function to compare with and make suggestions for.	
autocomplete for ontology class	Specifies the ontology class that provides the basis for the autocomplete function to compare with and make suggestions for.	
execution step triggered	Specifies the execution step that is triggered.	
has default placeholder value	This annotation property specifies a certain value (usually a literal) that is supposed to be used as the default placeholder value/resource selected by this entry component. This value/resource is depicted until the user selects a resource.	
has GUI input type	This annotation property specifies the type of GUI input required for triggering the execution step specified in 'execution step triggered' annotation property.	
include information from execution steps of individual [BOOLEAN]	This annotation property specifies, whether the ongoing execution process must include the information of execution steps defined in the corresponding individual resource. If value 'true', the execution process must include all information about execution steps from both class and individual resource, thereby merging the information to a single process description. Default value = 'false'	
input definition value defines SPrO variable resource	This annotation property specifies a specific variable carrying resource (i.e. SPrO variable resource), to which the input definition value (i.e. the definition of the selected resource) must be associated. As a consequence, if in a subsequent step this variable carrying resource is referenced, it functions as a placeholder for this input definition value.	
input label value defines SPrO variable resource	This annotation property specifies a specific variable carrying resource (i.e. SPrO variable resource), to which the input label value (i.e. the label of the selected resource) must be associated. As a consequence, if in a subsequent step this variable carrying resource is referenced, it functions as a placeholder for the input label value.	
input value/resource defines SPrO variable resource	This annotation property specifies a specific variable carrying resource (i.e. SPrO variable resource), to which the input value/resource (i.e. the value provided through user input or the URI of a selected resource) must be associated. As a consequence, if in a subsequent step this variable carrying resource is referenced, it functions as a placeholder for this input value/resource.	
not editable [BOOLEAN]	The value 'true' specifies that this entry component is not editable (e.g. a checkbox would be displayed and have the functionality of an inactive checkbox). In other words, this entry component provides no possibility to interact with it. Default value = 'false'	
requirement for triggering the execution step	This specifies further requirement(s) for an execution step to be triggered in addition to a change or input made by a user. Usually, a specific entry status is specified or a role or right, indicating that the execution step to be triggered can only be conducted for entries that possess the specified status or by a user who possesses the right or role.	
triggers 'click' for entry component	This annotation property specifies an entry component, for which a user input of type 'click' is automatically triggered.	
execution step: application operation	**This annotation triggers a specific defined operation in the middleware of the application.**	**executionStepApplicationOperation**
application operation 'redirect to hyperlink'	Specifies a URI to which the application will navigate.	
application operation 'save in cookie as key'	This annotation specifies a specific SPrO variable resource that must be saved in a cookie along with the resource that has been specified in the same execution step using the annotation 'application operation: save individual in cookie'.	
application operation 'save individual in cookie'	This annotation specifies a specific resource that must be saved in a cookie.	
subsequently redirected [BOOLEAN]'	If 'true', this annotation specifies that the application operation specified in this execution step must be triggered subsequently, after the ongoing execution step has been executed. Default value: 'false'	
execution step: copy and save triple statement(s)	**Specifies through respective annotations within this annotation all information required for copying triple statement(s) of one or more named graphs into memory and either write them directly to other named graphs or update them in memory and write the updated triple statement(s) to the source named graph or another named graph.**	**executionStepCopyAndSaveTripleStatements**
copied resource (of	Specifies either directly the resource to be identified or indirectly the class of the resource that must be iden-	

class) to be identified [input_*X*]	tified. The resource belongs to the copied resources. The variable carrying resource 'SPrO_VARIABLE: identified resource(s) [input_*X*]' refers to the resource identified based on this annotation. [with '*X*' being any letter between A and Z]	
copy all individuals of class	Specifies a specific class of which all individuals must be copied. This step includes copying all property assertions and annotations of these individuals.	
copy from named graph (of class)	Specifies the named graph (sometimes by specifying the class of named graph it belongs to) from which the triple statements are copied.	
copy from workspace	Specifies the workspace from which the triple statements are copied. The named graphs of which the content is copied are specified in another annotation.	
copy individual	Specifies a specific individual that must be copied. This step includes copying all property assertions and annotations of this individual.	
do not update URI of	This annotation property specifies a resource of which the URI must NOT be updated.	
exclude all triple statements with property	Specifies a property and all triple statements with the same property will not be copied.	
replace with new individual resources for	This annotation specifies a specific class (including all its subclasses) of individual resources for which new resources must be generated when copied from the named graph specified by an accompanying annotation property. E.g. when the class 'entry component' is specified, all resources referring to individual entry components must not be copied, but instead new individual resources generated that belong to the same classes and that share all properties with the resources to be "copied" - in other words: generate duplicates with the same properties but their own unique URL.	
update all URIs that share namespace with	Specifies a particular resource. All URIs of and in the copied named graphs that share the same namespace as the here specified resource must be updated with the new namespace specified in another accompanying annotation.	
update URIs using namespace of entry ID	This annotation property specifies the entry ID from which the namespace must be taken for updating the URIs.	
update composition [BOOLEAN]	If 'true', the composition specified in an accompanying triple statement must be updated after the execution process is terminated. Default value: 'false'	
and all subcommands from table 2		
execution step: copy named graphs	**Specifies through annotations within this annotation the named graphs to be copied, from which workspace they must be copied and to which namespace their URIs must be changed.**	**executionStepCopyNamedGraphs**
copy from workspace	Specifies the workspace from which the triple statements are copied. The named graphs of which the content is copied are specified in another annotation.	
do not update URI of	This annotation property specifies a resource of which the URI must NOT be updated.	
save new URIs to named graph	This annotation property specifies a named graph to which all newly created URIs must be stored to with their type specification.	
save new URIs to workspace	Specifies a particular workspace to which the newly generated URIs must be stored.	
update all URIs that share namespace with	Specifies a particular resource. All URIs of and in the copied named graphs that share the same namespace as the here specified resource must be updated with the new namespace specified in another accompanying annotation.	
update URIs of and in named graphs using namespace of entry ID	This annotation property specifies the entry ID from which the namespace must be taken for updating the URIs of and in the copied named graphs.	
update URIs using namespace of entry ID	This annotation property specifies the entry ID from which the namespace must be taken for updating the URIs.	
use ontology IDspace mapping	Specifies a SPrO variable carrying resource that contains the mapping between ontology IDspaces and their corresponding namespaces	
and all subcommands from table 2		
execution step: decision dialogue	**This annotation property is used to prompt messages through pop-up windows and the like and to communicate decisions that the user must make. Messages and decisions (=requests) are communicated using further annotation properties.**	**executionStepDecisionDialogue**
application dialogue-message	This annotation property is used to prompt a dialogue message through a pop-up window that requires some input from the user to be closed again.	
application error-message	This annotation property is used to prompt an error message through a pop-up window or the like to inform the user about incorrect input.	
application info-message	This annotation property is used to prompt a message through a pop-up window or the like to provide user with relevant information.	
end action operation	This annotation property is used for indicating the end of an execution step chain by triggering the specified SPrO operation. When reaching this step, the action is executed, the specified operation triggered and all subsequent execution steps must be ignored. This annotation is often used in combination with an 'execution	

	step: if-then-else statement' annotation, which can result in a bifurcation of the execution chain. Allowed values are restricted to individuals of the class 'SPrO operation'.	
SPrO OPERATION: end action	*This SPrO operation specifies that the currently executed list of execution steps ends, irrespective of any non-executed steps still remaining in the list.*	
SPrO OPERATION: ERROR end action	*This SPrO operation is used in cases in which a sequence of execution steps is stopped because of a wrong input.*	
execution step: define variables	**This annotation property is used to define SPrO variable resource values that will be used in a subsequent execution step.**	**executionStepDefineVariables**
add resource to list	This annotation property specifies a list to which a resource must be added that has been specified in an accompanying triple statement.	
delete resource from list	This annotation property specifies a list from which a resource must be deleted that has been specified in an accompanying triple statement.	
resource to be added to a list	This annotation property specifies a specific resource that must be added to a list of resources. The list-resource is specified in an accompanying triple statement.	
resource to be deleted from a list	This annotation property specifies a specific resource that must be deleted from a list of resources. The list-resource is specified in an accompanying triple statement.	
resource(s) to be identified [input_*X*]	Specifies either directly the resource to be identified or indirectly the class of the resource that must be identified. The variable carrying resource 'SPrO_VARIABLE: identified resource(s) [input_*X*]' refers to the resource identified based on this annotation. [with '*X*' being any number between 1 and 30]	
execution step: delete all triple statements of named graph	**Specifies through annotations within this annotation the named graphs of which all triple statements must be deleted, together with the workspace of this named graph.**	**executionStepDeleteAllTriplesOfNamedGraph**
and all subcommands from table 2		
execution step: delete multiple triple statements	**Specifies through respective annotations within this annotation which triple statements must be deleted and from which named graph and which work space.**	**executionStepDeleteMultipleTripleStatements**
and all subcommands from table 2		
and all subcommands from 'execution step: save/delete triple statement(s)'		
execution step: delete named graphs	**Specifies through annotations within this annotation the named graphs to be deleted, from which workspace they must be deleted and to which core ID or entry ID they belong to.**	**executionStepDeleteNamedGraphs**
and all subcommands from table 2		
execution step: delete part of composition	**Specifies through annotations within this annotation a particular entry component. All triple statements that have this entry component or one of its children entry components as a subject or an object must be deleted from the specified named graph and workspace.**	**executionStepDeletePartOfComposition**
delete entry component with all of its children	Specifies a particular entry component. All triple statements that have this entry component or one of its children entry components as a subject or an object must be deleted from the named graph and workspace that are specified in accompanying annotations. This must be conducted to the entire hierarchy of children.	
delete entry component with all of its children (list)	Specifies a list of particular entry components. All triple statements that have one of these entry components or one of its children entry components as a subject or an object must be deleted from the named graph and workspace that are specified in accompanying annotations. This must be conducted to the entire hierarchy of children for each entry component in the list.	
update composition [BOOLEAN]	If 'true', the composition specified in an accompanying triple statement must be updated after the execution process is terminated. Default value: 'false'	
and all subcommands from table 2		
execution step: execute now	**This annotation triggers the processing of execution steps and input held in memory that was provided through specific workflow actions or entry components that have the annotation 'wait for workflow execution [BOOLEAN]' 'true' or 'wait for execution [BOOLEAN]' 'true'.**	**executionStepExecuteNow**
execute now [BOOLEAN]	For the value 'true', this annotation specifies that all input held in memory due to the 'wait for execution [BOOLEAN]' annotation property is used in the running action. Default value: 'false'	
update store [BOOLEAN]	If 'true', this annotation specifies that all changes held in memory must be processed and the store updated for the specified named graphs. Default value: 'false'	
wait for execution [BOOLEAN]	For the value 'true', this annotation specifies that all input given through this entry component must be held in memory-queue until the 'execution step: execute now' or 'execute now [BOOLEAN]' is triggered, in which case all input held in memory must be processed in order. Default value: 'false'	
execution step: extract and save entry composition	**Specifies through annotations within this annotation the root element of a composition, which can be used as a starting point for extracting the corresponding composition from the application ontology. Further annotation properties specify the named graph to which the composition must be saved and the work-**	**executionStepExtractAndSaveEntryComposition**

	space of this named graph.	
composition has root element	Specifies the root element of a composition, which can be used as a starting point for extracting the corresponding composition from the application ontology. Further annotation properties specify the named graph to which the composition must be saved and the workspace of this named graph.	
and all subcommands from table 2		
execution step: generate resources	**This annotation property specifies through respective annotations within annotations a list of individual resources that must be newly generated. These will be used in various triple statements throughout this workflow action.**	**executionStepGenerateResources**
change version ID to status	This annotation property specifies the new entry status of the entry for which the resources are generated.	
generate resource of class [input_*X*]	Specifies the class for the resource that must be generated. This implicitly requires the generation of the following triple statement: 'S:resource P:rdf:type O:class'. This triple statement must be added to the named graph specified in another annotation. The variable carrying resource 'SPrO_VARIABLE: generated resource [input_*X*]' refers to the resource generated based on this annotation. [with '*X*' being any number between 1 and 70]	
generates resources for entry ID	This specifies the entry for which the resources are generated.	
and all subcommands from table 2		
execution step: get DOI	**This annotation property triggers a procedure with which the entry that is going to be published will receive its DOI.**	**executionStepGetDOI**
DOI defines SPrO variable resource	This annotation property specifies a specific SPrO variable resource, to which the DOI that has been received during this execution step must be associated. As a consequence, if in a subsequent step this variable carrying resource is referenced, it is a place holder for this DOI, which must be used in this subsequent step instead of the SPrO variable resource .	
execution step: go to execution step	**This annotation property specifies through the 'go to execution step' annotation within this annotation the execution step that must be processed next.**	*This is a special case, because the associated subcommand can be used in any execution step*
go to execution step	This annotation property specifies the execution step that must be processed next.	
execution step: hyperlink	**Specifies through respective annotations within this annotation a page to which the application should navigate to. The overall composition of the page may have been specified through previous execution steps ('execution step: specifications and allocations for hyperlink').**	**executionStepHyperlink**
is general application page [BOOLEAN]	This annotation property specifies a static page within the application that is not entry/input-data specific. The default value is 'false'.	
SPrO variable value transferred to hyperlink	This annotation specifies a SPrO variable resource with a specific value, which was assigned during the current workflow action. This value must be transferred to the here specified hyperlink.	
switch to entry	This annotation property specifies an entry ID, which the application uses for representing the composition specified in a previous 'execution step: specifications and allocations for hyperlink' in combination with an overlay or a page.	
switch to overlay	This annotation property specifies a widget, which the application uses for representing the composition specified in a previous 'execution step: specifications and allocations for hyperlink' in an overlay.	
switch to page	This annotation property specifies a widget, which the application uses for representing the composition specified in a previous 'execution step: specifications and allocations for hyperlink' in a page.	
update composition [BOOLEAN]	If 'true', the composition specified in an accompanying triple statement must be updated after the execution process is terminated. Default value: 'false'	
execution step: if-then-else statement	**Specifies through respective annotations within this annotation a certain Boolean condition ('if:') that triggers a certain consequence ('then:') if true and an alternative ('else:') if false. Often, however, an alternative is not specified, in which case no consequence results from the condition being not met and the process continues with the next execution step.**	**executionStepIfThenElseStatement**
else:	Specifies an alternative to a consequence described in another annotation ('then:'), which is triggered if a Boolean condition that is described in yet another annotation ('if:') is true. This alternative is triggered if the Boolean condition is false.	
has IF input value	This annotation property specifies an input value for an IF statement.	
has IF operation	This annotation property specifies the operation of an IF statement.	
has IF target value	This annotation property specifies an input target value for an IF statement.	
has THEN operation	This annotation property specifies the operation of a THEN statement.	
then:	Specifies a consequence ('then:') that is triggered if the Boolean condition described in another annotation ('if:') is true. If the resource connected to this annotation as an 'object' is a specific workflow action, this action must be triggered in case the Boolean condition is true.	
SPrO_VARIABLE:	*This SPrO variable resource is used whenever the general data scheme requires a specific variable to be*	

empty	*added, but for this specific case the variable is not needed and thus remains empty. If in subsequent execution steps this SPrO variable resource is used, the corresponding steps must not be executed but skipped.*
SPrO IF OPERATION: ALL empty	*This SPrO IF operation checks whether the SPrO variable resources specified in accompanying 'has IF input value' annotation(s) have some resource assigned to them or whether they are empty. In case they are ALL empty, the IF statement is true (and the 'then:' statement must be processed). If at least one of them has a specific resource assigned to it, the IF statement is false (and the 'else:' statement must be processed).*
SPrO IF OPERATION: ALL input already exists in triple store	*This SPrO IF operation checks whether the (SPrO variable) resources specified in accompanying 'has IF input value' annotations ALL are new to the triple store. This may include accompanying 'has IF target value' annotations that specify relevant class resources to which the searches of the application will be restricted. In case ALL of the specified resources already exist in the triple store, the IF statement is true (and the 'then:' statement must be processed). If at least one resource is new to the triple store, the IF statement is false (and the 'else:' statement must be processed).*
SPrO IF OPERATION: ALL input equal	*This SPrO IF operation checks whether the SPrO variable resources specified in accompanying 'has IF input value' annotations ALL have the same resource assigned to them. In case they ALL have the same resource assigned to them, the IF statement is true (and the 'then:' statement must be processed). If at least one of them has another resource assigned to it, the IF statement is false (and the 'else:' statement must be processed).*
SPrO IF OPERATION: ALL input is of target type	*This SPrO IF operation checks whether the (SPrO variable) resources specified in accompanying 'has IF input value' annotations ALL are of the class resource specified in the accompanying 'has IF target value' annotation. In case they ALL represent individuals of this class resource, the IF statement is true (and the 'then:' statement must be processed). If at least one of them has no resource assigned to it or the assigned resource is not an individual of this class (or one of its subclasses), the IF statement is false (and the 'else:' statement must be processed).*
SPrO IF OPERATION: ALL input is some resource	*This SPrO IF operation checks whether the SPrO variable resources specified in accompanying 'has IF input value' annotations ALL have some resource/URI assigned to them (as opposed to some label/value). In case they ALL have some resource in the form of a URI assigned to them, the IF statement is true (and the 'then:' statement must be processed). If at least one of them has a label or value assigned to it, the IF statement is false (and the 'else:' statement must be processed).*
SPrO IF OPERATION: ALL input is value	*This SPrO IF operation checks whether the SPrO variable resources specified in accompanying 'has IF input value' annotations ALL have some label or value assigned to them (as opposed to some resource/URI). In case they ALL have some label or value assigned to them, the IF statement is true (and the 'then:' statement must be processed). If at least one of them has a resource/URI assigned to it, the IF statement is false (and the 'else:' statement must be processed).*
SPrO IF OPERATION: ALL null	*This SPrO IF operation checks whether the SPrO variable resources specified in accompanying 'has IF input value' annotation(s) are known to the application. In case they are ALL not known, the IF statement is true (and the 'then:' statement must be processed). If at least one of them is known to the application, the IF statement is false (and the 'else:' statement must be processed).*
SPrO IF OPERATION: log in check	*This SPrO IF operation checks whether the SPrO variable resources specified in accompanying 'has IF input value' annotation(s) contain the correct log in information of one of the users of the application (user email + user password). If correct, it is treated as 'true' and the specified 'THEN' clause must be executed. If incorrect, it is treated as 'false' and the specified 'ELSE' clause must be executed.*
SPrO IF OPERATION: SOME empty	*This SPrO IF operation checks whether the SPrO variable resources specified in accompanying 'has IF input value' annotation(s) have some resource assigned to them or whether they are empty. In case at least one (=SOME) is empty, the IF statement is true (and the 'then:' statement must be processed). If all of them have a specific resource assigned to it, the IF statement is false (and the 'else:' statement must be processed).*
SPrO IF OPERATION: SOME equal	*This SPrO IF operation checks whether the SPrO variable resources specified in accompanying 'has IF input value' annotations have the same resource assigned to them. In case at least two (=SOME) have the same resource assigned to them, the IF statement is true (and the 'then:' statement must be processed). If all of them have a different resource assigned to them, the IF statement is false (and the 'else:' statement must be processed).*
SPrO IF OPERATION: SOME input already exists in triple store	*This SPrO IF operation checks whether the (SPrO variable) resources specified in accompanying 'has IF input value' annotations are new to the triple store. This may include accompanying 'has IF target value' annotations that specify relevant class resources to which the searches of the application will be restricted. In case at least one (=SOME) of the specified resources already exists in the triple store, the IF statement is true (and the 'then:' statement must be processed). If all resources are new to the triple store, the IF statement is false (and the 'else:' statement must be processed).*
SPrO IF OPERATION: SOME input is of target type	*This SPrO IF operation checks whether the (SPrO variable) resources specified in accompanying 'has IF input value' annotations are of the class resource specified in the accompanying 'has IF target value' annotation. In case at least one (=SOME) resource represents an individual of this class resource, the IF statement is true (and the 'then:' statement must be processed). If all of them have no resource assigned to them or if all have a resource assigned that is not an individual of this class (or one of its subclasses), the IF statement is false (and the 'else:' statement must be processed).*
SPrO IF OPERATION: SOME input larger than target	*This SPrO IF operation checks whether some value specified in accompanying 'has IF input value' annotation(s) is higher than the value specified in the accompanying 'has IF target value' annotation. In case at least one input value is larger, the IF statement is true (and the 'then:' statement must be processed).*

SPrO IF OPERATION: SOME null	*This SPrO IF operation checks whether the SPrO variable resources specified in accompanying 'has IF input value' annotation(s) are known to the application. In case at least one (=SOME) is not known, the IF statement is true (and the 'then:' statement must be processed). If all of them are known to the application, the IF statement is false (and the 'else:' statement must be processed).*	
SPrO THEN OPERATION: subsequent execution steps with 'delete triple statement [BOOLEAN]' = 'true'	*This SPrO THEN operation specifies that for all subsequent execution steps of the type 'execution step: save/delete triple statement(s)' the specified triple statements must be deleted and not saved. This is the same as if these execution steps would be annotated with 'delete triple statements [BOOLEAN]' = 'true'.*	
execution step: save/delete triple statement(s)	**Specifies through respective annotations within this annotation the resources and values of the triple statements that i) must be generated by the application or ii) must be deleted. Additional annotations specify the named graph and workspace where the triple statement must be saved to or deleted from.**	**executionStepSaveDeleteTripleStatements**
object	This annotation property is used in combination with the 'subject' and 'property' annotation property to describe a triple statement. It specifies the resource/value that takes the 'object' position within the triple statement.	
object (all individuals of class)	Same as the annotation 'object', with the difference that the range is any individual of the specified class.	
object (copied individual of)	Same as the annotation 'object', with the difference that the range is an individual of the specified class that has been copied in a previous execution step within this workflow action.	
object (this entry's specific individual in memory of)	Same as the annotation 'object', with the difference that the range is an individual of the specified class that refers to the entry ID that is currently in focus (the one to which the variable carrying resource 'SPrO_VARIABLE: this entry ID' refers to) AND the individual is currently held in memory. This is for instance used when referring to a specific entry component of which several individuals exist in a given composition.	
object (this entry's specific individual of)	Same as the annotation 'object', with the difference that the range is an individual of the specified class that refers to the entry ID that is currently in focus (the one to which the variable carrying resource 'SPrO_VARIABLE: this entry ID' refers to). This is for instance used when referring to a specific entry component of which several individuals exist in a given composition.	
object (unique individual of)	Same as the annotation 'object', with the difference that the range is an individual of the specified class that is the only individual of that class in the named graph that is specified in an accompanying triple statement.	
object list	Same as the annotation 'object', with the difference that the range is a list of individual resources.	
an analogous list of annotation properties for 'subject'		
property	This annotation property is used in combination with the 'object' and 'subject' annotation property to describe a triple statement. It specifies the property within the triple statement.	
delete triple statement [BOOLEAN]	The default value is 'false'. The value 'true' indicates that the respective triple statement must be deleted. Specifies through respective annotations within this annotation the resources and values of a triple statement that must be deleted.	
and all subcommands from table 2		
execution step: search triple store	**This annotation triggers a search of the triple store for the existence of a specific resource. Further annotations specify what is being searched and specify criteria that narrow in the search space.**	**executionStepSearchTripleStore**
multiple-hits-search [BOOLEAN]	The value 'true' indicates that the respective search may yield more than one hit. The default value is 'false'.	
search restricted to entry URI (except for terminal counter value)	This annotation property specifies an entry ID. The result of the search that is specified in accompanying annotation properties is restricted to resources that share the same URI as the here specified entry ID, with the exception that their last value may differ (i.e. the counter value that follows the last "_").	
search target	Specifies through the SPrO variable resource 'SPrO_VARIABLE: subject', 'SPrO_VARIABLE: object', 'SPrO_VARIABLE: property' or 'SPrO_VARIABLE: named graph' as its object, what element of the triple statement to be searched is the target of this search.	
search target defines SPrO variable	Specifies the SPrO variable under which the search target result is referenced.	
search target saved to 'list of URIs named graph' SPrO variable	This annotation property triggers that the resource(s) identified through the search (in case 'multiple-hits-search [BOOLEAN]='true', this can be more than one resource) will be stored into an internal hash map that is identified through the specified SPrO variable resource.	
use only results based on namespace for	This annotation property specifies an entry status. Only the results of the search of which the URIs match the here specified status will be considered and assigned to the target SPrO variable. If several of these annotation properties are used in the same execution step, all the therein specified stati must be considered.	
wild card search [BOOLEAN]	The value 'true' indicates that the respective search uses a wild card function in case literals are searched. The default value is 'false'.	

and all subcommands from 'execution step: save/delete triple statement(s)'		
and all subcommands from table 2		
SPrO_VARIABLE: ?	*This SPrO variable is used for indicating an unknown resource in a triple statement that, for instance, has to be SPARQLed or that can vary for the task to be conducted.*	
SPrO_VARIABLE: object	*used in combination with the 'search target' subcommand*	
SPrO_VARIABLE: subject	*used in combination with the 'search target' subcommand*	
SPrO_VARIABLE: property	*used in combination with the 'search target' subcommand*	
execution step: specifications and allocations for hyperlink	**This annotation property is used to prepare a subsequent 'execution step: hyperlink'. It specifies and allocates the composition(s) and components used in the hyperlink.**	**executionStepSpecificationsAndAllocationsFor Hyperlink**
position	Specifies the position of this entry component in its parent component or widget.	
use entry component	This annotation property specifies an entry component that is used in a later 'execution step: hyperlink'.	
use hierarchy/composition from entry	This annotation property specifies an entry ID of a composition that is used in a later 'execution step: hyperlink'.	
use root element	This annotation property specifies a root element of some hierarchy that is used in a later 'execution step: hyperlink'.	
use tab	This annotation property specifies a tab from the interface that is used in a later 'execution step: hyperlink'.	
use union of compositions with child root entry component	This annotation property specifies that a union of compositions must be created that is used in a later 'execution step: hyperlink'. The entry component specified through this annotation property is the root element of a composition that must be added to the composition of the root element specified through the annotation property 'use union of compositions with parent root entry component'.	
use union of compositions with parent root entry component	This annotation property specifies that a union of compositions must be created that is used in a later 'execution step: hyperlink'. The entry component specified through this annotation property is the overall root element of this merged union of compositions. Through the annotation property 'use union of compositions with child root entry component', further compositions are specified through their respective root elements. These additional compositions must be added to the composition of the parent root element.	
and all subcommands from table 2		
execution step: trigger workflow action	**This annotation property is used as a wrapper in combination with other annotation properties that specify all information required for triggering a specific workflow action.**	**executionStepTriggerWorkflowAction**
requirement for triggering a workflow action	This specifies further requirement(s) for an action to be triggered in addition to a change or input made by a user. Usually, a specific entry status is specified, indicating that the action to be triggered can only be conducted for entries that possess the specified status. In case more than one requirement is specified, the relation between the requirements is OR not AND!	
SPrO variable value transferred to triggered action	This annotation specifies a SPrO variable resource that has a specific value, which was assigned during the current workflow action. This value must be transferred to the here triggered workflow action. E.g.: if the variable carrying resource 'SPrO_VARIABLE: identified resource(s) [input_2]' has a specific individual resource assigned, it must have the same resource assigned in the triggered workflow action.	
subsequently triggered workflow action [BOOLEAN]	If 'true', this annotation specifies that the workflow action triggered in this execution step must be triggered subsequently, after the ongoing execution step has been executed. Default value: 'false'	
trigger action of button (of class)	This annotation specifies an entry component that is a button, of which the action must be executed. It is usually used in combination with the annotation 'subsequently triggered workflow action [BOOLEAN]'.	
triggers workflow action	This annotation specifies a workflow action that must be triggered for an entry ID that is specified in another annotation.	
execution step: update triple statement(s)	**Specifies through respective annotations within this annotation all information required for changing a specific resource in all triple statements of a specified named graph.**	**executionStepUpdateTripleStatements**
to be updated resource/value	Specifies a particular resource or value that must be changed in all the named graphs that are specified in accompanying annotations.	
to be updated resource/value (individual of)	Same as the annotation 'to be updated resource/value', with the difference that the range is an individual of the specified class.	
update with (copied individual of)	Same as the annotation 'update with resource/value', with the difference that the range is an individual of the specified class that has been copied in a previous execution step within this workflow action.	
update with (this entry's specific individual of)	Same as the annotation 'update with resource/value', with the difference that the range is an individual of the specified class that refers to the entry ID that is currently in focus (the one to which the variable carrying resource 'SPrO_VARIABLE: this entry ID' refers to). Contrary to the annotation 'update with resource/value', this annotation has no values as ranges.	
update only for values	This annotation property is used in combination with the 'update with resource/value' annotation property. It	

higher than	specifies a filter for what may be updated. Only triple statements with a value higher than here specified will be updated.
update with (unique individual of)	Same as the annotation 'update with resource/value', with the difference that the range is an individual of the specified class that is the only individual of that class in the named graph that is specified in an accompanying triple statement.
update with resource/value	Specifies the particular resource or value with which the to-be-changed resource/value is replaced. Which resources/values must be changed is specified in accompanying annotations.
and all subcommands from 'execution step: save/delete triple statement(s)'	
and all subcommands from table 2	
SPrO_VARIABLE: to be updated	*This SPrO variable is used to indicate which element (subject, property, object) in a triple statement is subject to an update.*

Table 1: Excerpt of SPrO annotation properties that are used as execution step commands and their corresponding Java method (light gray background) as well as the SPrO annotation properties that are used as their associated subcommands (white background) and, if applicable, the defined value resources (SPrO variables, shown in italics). The first execution step (dark grey background) is a special case for the middleware because input from and output for the web socket are treated differently than in the case of the other commands and subcommands.

SPrO annotation properties for subcommands specifying named graph and workspace	**Description**
load from/save to/update in all named graphs (this entry's specific individual of) of this SPrO variable list	Same as the annotation 'load from/save to/update in named graph (this entry's specific individual of)', with the difference that the range is a list of named graph classes from which the individuals that refer to the entry ID that is currently in focus must be inferred.
load from/save to/update in multiple named graphs (this entry's specific individuals of)	Same as the annotation 'load from/save to/update in named graph (individual of)', with the difference that the range refers to all individuals of the specified class that refer to the entry ID that is currently in focus.
load from/save to/update in named graph	Specifies the named graph from which a triple statement is loaded or to which it is saved or in which triple statements must be updated. Together with information about the entry type and the workspace, the directory can be located in the triple store for saving or loading the triple statement(s).
load from/save to/update in named graph (copied individual of)	Specifies a subclass of the class named graph, the individual of which is the named graph from which a triple statement is loaded or to which it is saved or in which triple statements must be updated. The range is an individual of the specified class that has been copied in a previous execution step within this workflow action.
load from/save to/update in named graph (this entry's specific individual of)	Same as the annotation 'load from/save to/update in named graph', with the difference that the range is an individual of the specified class that refers to the entry ID that is currently in focus.
load from/save to/update in named graphs of this SPrO variable list	Same as the annotation 'load from/save to/update in named graph', with the difference that the range is a list of named graph individuals.
named graph belongs to entry ID	Specifies a particular entry ID. The respective triple statement(s) to which this annotation property relates to must be stored to or loaded from a named graph that belongs to this entry ID.
named graph belongs to workspace	Specifies a particular workspace. The respective triple statement(s) to which this annotation property relates to must be stored to or loaded from this respective workspace.
set new focus on entry ID	This annotation specifies an entry ID that takes the function of the entry currently in focus for the ongoing action. In other words, the SPrO variable resource 'this entry ID' and all its associated SPrO variable resources will refer to the specified entry ID and its associated resources, just as if the action had been triggered with the specified entry ID in focus. This change in focus to the specified entry ID holds in all subsequent execution steps of this action, until another 'set new focus on entry ID' annotation changes the focus again. If used within an execution step, this annotation is executed first before any other annotations of this execution step can be executed.
set new focus on entry ID (individual of)	This annotation specifies a class of entry ID that defines the focus for the ongoing action. Based on the specified class and the point from which the respective action has been triggered, the middleware is able to identify the individual entry ID that defines the entry in focus. Based on this focus, the middleware is able to resolve all references to classes that are intended to be references to individual resources to the respective individual resources. This specified focus holds in all subsequent execution steps of this action, until another 'set new focus on entry ID' annotation changes the focus again. If used within an execution step, this annotation is executed first before any other annotations of this execution step can be executed.
set new focus on entry ID for this execution step	This annotation specifies an entry ID that takes the function of the entry currently in focus for this execution step only. In other words, the SPrO variable resource 'this entry ID' and all its associated SPrO variable resources will refer to the specified entry ID and its associated resources, just as if the action had been triggered with the specified entry ID in focus. This change in focus to the specified entry ID is restricted to this execution step and will return to the former focus when the execution step

	has been executed. If used within an execution step, this annotation is executed first before any other annotations of this execution step can be executed.

Table 2: SPrO annotation properties that are used as subcommands for specifying the location in the application's triple store framework for loading, saving and updating triple statements and for setting the focus on a specific entry ID.

SPrO annotation properties for GUI-related commands and relevant defined values (SPrO variables)	**Description**
component status [BOOLEAN]	This annotation property functions like a switch between two ways in which this entry component can be represented (status 'true' and status 'false').
drag-and-drop position enabled [BOOLEAN]	This annotation property specifies whether a user can drag and drop this entry component to change its position relative to other entry components that can be dragged and dropped. If 'true', this component's position can be changed through drag and drop.
drag-and-drop restricted to	This annotation property specifies within which entry component and thus which area of the GUI this entry component can be moved using drag and drop.
has associated instance resource [input_*X*]	Specifies an instance resource that is linked to this instance of an entry component class. This way, the resource is held available for the case it must be referred to in some other context. The SPrO variable resource 'SPrO_VARIABLE: associated instance resource [input_*X*]' refers to the input based on this annotation. [with '*X*' being any letter between A and N]
has associated 'list of URIs named graph' [input_*X*]	Specifies the instance of a particular named graph in which a set of URIs is listed. A particular SPrO variable resource 'SPrO_VARIABLE: associated 'list of URIs named graph [input_*X*]' refers to the list or URIs contained in the indicated named graph. [with '*X*' being any letter between A and T]
has Boolean value [BOOLEAN]	Assigns a Boolean value (or a variable carrying resource from which a Boolean value can be inferred) to this entry component, which is used for read-only purposes.
has default placeholder value	This annotation property specifies a certain value (usually a literal) that is supposed to be used as the default placeholder value/resource selected by this entry component. This value/resource is depicted until the user selects a resource.
has GUI representation	This annotation property specifies the HTML element with which this entry component is represented in the GUI.
has scrollbar	This annotation property specifies the type of scrollbar that this entry component should have.
hidden [BOOLEAN]	This annotation property specifies the visibility of a given entry component. If 'true', the respective entry component is not hidden, if 'false', it is visible.
hyperlink	Specifies a hyperlink to this resource.
input of type [info-input_*X*]	Specifies some information that is important for an input (e.g. the type of entry that will be newly created when pushing a 'create new entry' button). The SPrO variable resource 'SPrO_VARIABLE: input of type [info-input_*X*]' refers to the input based on this annotation. [with '*X*' being any number between 1 and 20]
label status 'false'	This specifies the content of a visible label for the status 'false' of this entry component. This annotation is accompanied by the annotation property 'component status [BOOLEAN]' that specifies the status ('true' or 'false') of this entry component. The label is only visible if the component has the component status 'false'.
label *X*	This specifies the content of a visible label. Label 1 is the first label (from left to right) and further labels may exist. [with '*X*' being any number between 1 and 5]
required input [BOOLEAN]	Specifies, whether the user must provide input for this entry component because this information is required for the respective type of entry. IMPORTANT NOTE: If some input has been provided for an entry component with this annotation, a user cannot delete this input anymore, but merely change it; some valid input must always remain, once input has been provided.
resource(s) to be identified [input_*X*]	Specifies either directly the resource to be identified or indirectly the class of the resource that must be identified. Which resource(s) should be identified is further narrowed in by specifying a triple statement in which the resource is used. The SPrO variable resource 'SPrO_VARIABLE: identified resource(s) [input_*X*]' refers to the resource identified based on this annotation. [with '*X*' being any number between 1 and 30]
tooltip text	This specifies the content of a tooltip text, which appears when the mouse hovers over this entry component.
triggers 'click on' (individual of)	This annotation property specifies an individual entry component for which a 'click on' event is triggered when this button is clicked.
with information text	This specifies a short text phrase that is shown in a text input field (e.g., "enter label"; "enter URL"), in order to provide information about what should be entered in this text field. This text is not stored as input data and disappears immediately when the input field is selected.
SPrO_VARIABLE: input of type [info-input_X]	*This SPrO variable refers to the information/resource for which the annotation 'input of type [info-input_X]' specifies the input-information.* [with '*X*' being any number between 1 and 20]
SPrO_VARIABLE: identified resource(s) [input_X]	*This SPrO variable refers to the resource(s) that has been identified based on the annotation 'resource(s) to be identified [input_X]' that has been used in some earlier execution step within this action. If several execution steps of this action have used this annotation, the SPrO variable refers to the last usage.* [with '*X*' being any number between 1 and 30]
SPrO_VARIABLE: associated instance resource	*This SPrO variable refers to the resource specified by the annotation property 'has associated instance resource [input_X]' that links it to an entry component. This way, the resource is held available for the case that*

[input_X]	*it must be referred to in some other context.* [with '*X*' being any letter between A and N]
SPrO_VARIABLE: associated 'list of URIs named graph' [input_X]	*This SPrO variable refers to a particular named graph that contains a list of URIs which has been specified by the annotation property 'has associated 'list of URIs named graph' [input_X]'. Whenever the SPrO variable is used in an execution step as placeholder for a resource, all resources (URIs) contained in the specified named graph must be processed consecutively before proceeding with the next execution step.* [with '*X*' being any letter between A and T]
Relevant SPrO object properties	**Description**
belongs to radio button group	Specifies to which GUI radio button group this radio button entry component belongs to.
entry component of	Specifies an entry component that this entry component is part of. The parthood relations between entry components describe the hierarchical encaptic structure in which the data belonging to a data entry are organized. The hierarchy of components nested within components forms the overall entry composition for a specific data entry, which at its turn describes the organization of data items into sets and subsets, resulting in a partonomic composition of groups of data items that are organized in entry components.
has entry component	Specifies an entry component that is part of this entry component. The parthood relations between entry components describe the hierarchical encaptic structure in which the data belonging to a data entry are organized. The hierarchy of components nested within components forms the overall entry composition for a specific data entry, which at its turn describes the organization of data items into sets and subsets, resulting in a partonomic composition of groups of data items that are organized in entry components.
has selected resource	This object property specifies a specific resource that is to be recorded as the resource that has been selected for this entry component. The label of this resource is depicted in the GUI until the user selects a different resource.
has user/GUI input [input_*X*]	Specifies a resource that is the user/GUI input that is based on one of the 'input restricted to individuals of [input_*X*]' annotations. It links the component resource via this object property to the resource that is the user/GUI input. This way one can easily document the input within the entry composition. [with '*X*' being any letter between A and G]
has user/GUI input [label]	Specifies a label or value that belongs to the user/GUI input.
has user/GUI input [URI]	Specifies a resource (i.e. URI) that belongs to the user/GUI input.
Relevant SPrO data properties	**Description**
has position in entry component	Specifies the position (in the order: from left to right and top to bottom) of an entry component within its parent entry component.
has user/GUI input [value_A]	Specifies a value that is the user/GUI input. It links the component resource via this data property to the value that is the user/GUI input. This way one can easily document the input within the entry composition.
has visible label *X*	A visible label is a string that is visible in the interface somewhere within the area of its GUI element. Label *X* is the *X*th label (from left to right). Further labels may exist. [with '*X*' being any number between 1 and 8]
new row [BOOLEAN]	If 'true', this data property specifies that this entry component must be positioned in a new row. Default value: 'false'

Table 3: SPrO annotation properties that are used as GUI-related commands together with relevant defined value resources (SPrO variables, shown in italics) as well as relevant SPrO object and SPrO data properties.

The location of a triple statement in the application's tuple store framework is therefore defined by the combination of workspace and named graph. Table 2 lists all SPrO annotation properties that can be used as subcommands within execution steps for specifying the location of triple statements.

Besides the specifications of database processes and other workflows, SPrO can also be used to specify the GUI of a data-centric Semantic Web application. This includes the description of HTML entry forms for sign up and log in as well as for user profiles, but also for all types of data entry forms used in the application.

Table 3 lists all the SPrO annotation properties that can be used as commands to describe the overall composition of an HTML page of a data entry. Each page is described as a set of entry components. Each entry component is represented in the corresponding SCO as an instantiated ontology class. A given page-composition is thus represented as a set of ontology classes, of which the instances are linked to each other through specific object properties (*'entry component of'* and *'has entry component'*) into parent-child relations, resulting in an encaptic hierarchy of ontology instances. The resulting instance-based semantic graph, which is described in the application's SCO, then functions as a template for the data entry form of a given type of data entry for the application. Each newly created data entry of a certain type is a copy of its respective template graph.

The position of a child entry component in relation to its sibling entry components is specified through a respective data property (*'has position in entry component'*) on the child itself. The HTML representation, as well as the input control and the specific functionality of each component, are specified in the corresponding SCO class. The class also specifies the data scheme of how the user input must be translated

into a semantic data graph and where this data graph must be stored in the Jena tuple store framework in terms of workspace and named graph.

The Java-based middleware interprets all these descriptions, produces the application and coordinates the application's overall operation based on the information from the descriptions in SCO. This includes interpreting the descriptions of data entry forms, data views, the overall architecture of the GUI and the actual data in the tuple store framework, communicating these interpretations with the frontend, interpreting the user input from the frontend and processing this input in accordance with the descriptions from SCO. The middleware thus mediates between SPrO, SCO and data graphs in the underlying Jena tuple store framework on the one hand and the browser-based GUI with the user input and user interaction on the other hand (Fig. 1).

3. SPrO Use Cases

We believe that one of the main reasons why most data-centric applications in science do not use semantic technology to its full potential can be found in the rather complex structure of semantic data graphs. Semantic data graphs are often bulky and to a high degree cross-linked, which makes them hard to comprehend for a human reader. If human readers have substantial problems with directly consuming semantic graphs, data-centric applications that store data and metadata as semantic graphs would have to translate those graphs into something that is more accessible, such as tables and data entry forms presented in HTML pages. As already mentioned above, SPARQL endpoints are no solution in this regard, as they only allow direct interaction with the graphs themselves.

So far, the development of data-centric Semantic Web applications that integrate semantic graphs with a user-friendly representation of their data has been hampered by a lack of application development frameworks that are well integrated with RDF/OWL. With SPrO and its accompanying Java-based middleware, we attempt to provide such a development framework and thus close the gap between the needs for a user-friendly representation and an eScience-compliant documentation of data and metadata.

3.1. Semantic Ontology-Controlled Application for Web Content Management Systems (SOCCOMAS)

We use SPrO for describing a semantic web-based content management system (S-WCMS), which we call SOCCOMAS [27,28]. Its source code ontology (available from [29]) contains descriptions of ready-to-use features and workflows typically required by an S-WCMS, including user administration with log-in and signup forms, user registration and login process and session management and user profiles. The SOCCOMAS source code ontology also defines a general publication life-cycle process for data entries that allows a user to create a revised draft version based on the current published version of a data entry. Every published data entry receives its own digital object identifier (DOI), and the creator of the entry can specify under which creative commons license the entry will be published. The publication life-cycle covers all transitions between the following possible states of a data entry:

1) current draft version;
2) backup draft version;
3) draft version in recycle bin;
4) deleted draft version;
5) current published version;
6) previously published version.

Moreover, the SOCCOMAS source code ontology specifies automatic procedures for tracking overall provenance (i.e., creator, authors, creation, and publication date, contributors, relation between different versions, etc.) for each particular data entry. On a significantly finer level of granularity, the source code ontology also specifies automatic procedures that track all changes made to a particular data record at the level of individual input fields and documents them in a detailed change-history log. This is especially useful when editing data entries collaboratively. All the gathered metadata are recorded in RDF following established data and metadata standards using terms and their corresponding URIs from established ontologies.

An S-WCMS run by SOCCOMAS (and thus by SPrO and its accompanying middleware) provides human-readable output in form of HTML and CSS for browser requests and access to a SPARQL endpoint for machine-readable service requests. Since every published entry has its own DOI and is published under a creative commons license, and since all data and metadata are documented as semantic graphs that are also accessible through a SPARQL endpoint, all data published of such a S-WCMS

reaches the five star rank of Tim Berners-Lee's rating system for Linked Open Data [30].

SOCCOMAS thus provides the description of all basic features and functionalities generally required for an S-WCMS. All specific features and functionalities needed for a particular S-WCMS, however, still have to be described in the source code ontology of that S-WCMS, including HTML templates for data entry forms, specifications of input control and overall behavior of each input field for the different types of data entries that the S-WCMS manages. These descriptions also include specifications of the underlying data scheme that determines how user input triggers the generation of data-scheme-compliant triple statements and where these triples must be saved in the Jena tuple store framework in terms of named graph and workspace.

3.2. Semantic Morph·D·Base

We use SOCCOMAS for developing a module for morphological descriptions for the morphological data repository Morph·D·Base [31]. This semantic version of Morph·D·Base utilizes the general functionality of an S-WCMS provided by SOCCOMAS, to which additional features specifically required for semantic Morph·D·Base have been added through its own source code ontologies (available from [32]). Semantic Morph·D·Base enables users to generate highly standardized and formalized morphological descriptions that are stored in the tuple store framework as instance-based semantic graphs. When describing an anatomical structure, users can reference any ontology class from any anatomy ontology that is available at BioPortal [33] and describe the structure and all of its parts as instances of these classes. Parts can be further described through defined data entry forms, often referencing specific ontology classes from PATO [34]. Semantic Morph·D·Base is currently still in development, but a prototype [35] can be accessed and functions as a proof of concept for SOCCOMAS and our semantic programming approach.

Using SOCCOMAS and semantic programming for developing the module for morphological descriptions has proven to save valuable resources and development time. The source code ontology for the semantic Morph·D·Base prototype has been written by a domain expert with knowledge in ontology engineering, but no expertise in any programming language. Moreover, because all the processes and functionalities that any S-WCMS requires such as login, signup, user administration, publication lifecycle, and various automatic tracking procedures are provided by SOCCOMAS, development of the prototype was restricted only to features specific to this prototype. Furthermore, the approach has also proven that changes to the organization of the GUI such as adding a new input field to a data entry form can be conducted on the fly, which facilitates a user-centered design approach to application development.

4. Conclusion

Semantic programming with SPrO and its accompanying middleware can help to close the gap between the need for user-friendly web applications and eScience-compliant data and metadata. Data-centric applications that are based on the here proposed semantic programming approach can provide HTML-based data views that are easily comprehensible to human users, thereby hiding the often rather complicated semantic graphs that represent the actual data. At the same time, data harvester services and other applications can readily consume the semantic graphs through the application's SPARQL endpoint.

Because all data and metadata of applications based on semantic programming are represented in form of semantic graphs that can be searched through the applications' SPARQL endpoint, because the semantic graphs link to ontology classes that provide semantic transparency for the concepts used in the data and metadata graphs, and because SPrO and its accompanying middleware enable automatic provenance tracking and detailed change-history tracking for all user input, data and metadata of these applications are maximally findable, accessible, interoperable and reusable, and thus comply with the FAIR guiding principles [2]. Moreover, they are also computer-parsable. Semantic programming with SPrO and its accompanying middleware would thus provide a means to fully utilize the potential of semantic technology for scientific data-centric Semantic Web applications.

Moreover, since not only data and metadata of respective applications are stored as semantic graphs, but the application's source code as an ontology, semantic programming stands for taking semantic transparency to a next level by also semantically describing the application itself using the terms defined in SPrO.

References

[1] J. Gray, Jim Gray on eScience: A Transformed Scientific Method, in: T. Hey, S. Tansley, and K. Tolle (Eds.), The Fourth Paradigm: Data-Intensive Scientific Discoveries, Microsoft Research, Redmond, Washington, 2009: pp. xvii–xxxi.

[2] M.D. Wilkinson, M. Dumontier, I.J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L.B. da Silva Santos, P.E. Bourne, J. Bouwman, A.J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C.T. Evelo, R. Finkers, A. Gonzalez-Beltran, A.J.G. Gray, P. Groth, C. Goble, J.S. Grethe, J. Heringa, P.A.. 't Hoen, R. Hooft, T. Kuhn, R. Kok, J. Kok, S.J. Lusher, M.E. Martone, A. Mons, A.L. Packer, B. Persson, P. Rocca-Serra, M. Roos, R. van Schaik, S.-A. Sansone, E. Schultes, T. Sengstag, T. Slater, G. Strawn, M.A. Swertz, M. Thompson, J. van der Lei, E. van Mulligen, J. Velterop, A. Waagmeester, P. Wittenburg, K. Wolstencroft, J. Zhao, and B. Mons, The FAIR Guiding Principles for scientific data management and stewardship, *Scientific Data*. 3 (2016) 160018. doi:10.1038/sdata.2016.18.

[3] L. Vogt, The future role of bio-ontologies for developing a general data standard in biology: chance and challenge for zoo-morphology, *Zoomorphology*. 128 (2009) 201–217. doi:10.1007/s00435-008-0081-5.

[4] L. Vogt, eScience and the need for data standards in the life sciences: in pursuit of objectivity rather than truth, *Systematics and Biodiversity*. 11 (2013) 257–270. doi:10.1080/14772000.2013.818588.

[5] L. Vogt, M. Nickel, R.A. Jenner, and A.R. Deans, The Need for Data Standards in Zoomorphology, *Journal of Morphology*. 274 (2013) 793–808. doi:10.1002/jmor.20138.

[6] A. Brazma, On the importance of standardisation in life sciences, *Bioinformatics*. 17 (2001) 113–114.

[7] A. Brazma, P. Hingamp, J. Quackenbush, G. Sherlock, P. Spellman, C. Stoeckert, J. Aach, W. Ansorge, C.A. Ball, H.C. Causton, T. Gaasterland, P. Glenisson, F.C.P. Holstege, I.F. Kim, V. Markowitz, J.C. Matese, H. Parkinson, A. Robinson, U. Sarkans, S. Schulze-Kremer, J. Stewart, R. Taylor, J. Vilo, and M. Vingron, Minimum information about a microarray experiment (MIAME)–toward standards for microarray data, *Nature Genetics*. 29 (2001) 365–371. doi:10.1038/ng1201-365.

[8] X. Wang, R. Gorlitsky, and J.S. Almeida, From XML to RDF: how semantic web technologies will change the design of "omic" standards, *Nature Biotechnology*. 23 (2005) 1099–1103.

[9] B. Smith, Ontology, in: L. Floridi (Ed.), Blackwell Guide to the Philosophy of Computing and Information, Blackwell Publishing, Oxford, 2003: pp. 155–166.

[10] S. Schulz, H. Stenzhorn, M. Boeker, and B. Smith, Strengths and limitations of formal ontologies in the biomedical domain, *RECIIS*. 3 (2009) 31–45. doi:10.3395/reciis.v3i1.241en.

[11] S. Schulz, and L. Jansen, Formal ontologies in biomedical knowledge representation., *IMIA Yearbook of Medical Informatics 2013*. 8 (2013) 132–46. http://www.ncbi.nlm.nih.gov/pubmed/23974561.

[12] L. Vogt, Morphological Descriptions in times of eScience: Instance-Based versus Class-Based Semantic Representations of Anatomy, n.d.

[13] M. Uschold, and M. Gruninger, Ontologies: Principles, Methods and Applications, *Knowledge Engineering Review*. 11 (1996) 39–136.

[14] S.-A. Sansone, P. Rocca-Serra, W. Tong, J. Fostel, N. Morrison, A.R. Jones, and R. Members, A Strategy Capitalizing on Synergies: The Reporting Structure for Biological Investigation (RSBI) Working Group, *OMICS: A Journal of Integrative Biology*. 10 (2006) 164–171.

[15] SPARQL Query Language for RDF. W3C Recommendation 15 January 2008, (n.d.). https://www.w3.org/TR/rdf-sparql-query/.

[16] OWL@Manchester: List of Reasoners, (n.d.). http://owl.cs.manchester.ac.uk/tools/list-of-reasoners/.

[17] K. Wenzel, KOMMA: An Application Framework for Ontology-based Software Systems, *Semantic Web Journal*. 0 (2010) 1–10. http://www.semantic-web-journal.net/sites/default/files/swj89_0.pdf.

[18] M. Buranarach, T. Supnithi, Y.M. Thein, T. Ruangrajitpakorn, T. Rattanasawad, K. Wongpatikaseree, A.O. Lim, Y. Tan, and A. Assawamakin, OAM: An Ontology Application Management Framework for Simplifying Ontology-Based Semantic Web Application Development, *International Journal of Software Engineering and Knowledge Engineering*. 26 (2016) 115–145. doi:10.1142/S0218194016500066.

[19] M.D. Wilkinson, B. Vandervalk, and L. McCarthy, The Semantic Automated Discovery and Integration (SADI) Web service Design-Pattern, API and Reference Implementation, *Journal of Biomedical Semantics*. 2 (2011) 8. doi:10.1186/2041-1480-2-8.

[20] D.D.G. Gessler, G.S. Schiltz, G.D. May, S. Avraham, C.D. Town, D. Grant, and R.T. Nelson, SSWAP: A Simple Semantic Web Architecture and Protocol for semantic web services, *BMC Bioinformatics*. 10 (2009) 309. doi:10.1186/1471-2105-10-309.

[21] D. Martin, M. Paolucci, S. Mcilraith, M. Burstein, D. Mcdermott, D. Mcguinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara, Bringing Semantics to Web Services: The OWL-S Approach, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. doi:10.1007/b105145.

[22] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. Van de Walle, RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data, *Proceedings of the 7th Workshop on Linked Data on the Web*. 1184 (2014). http://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf.

[23] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, and P. Colpaert, Triple Pattern Fragments: A low-cost knowledge graph interface for the Web, *Journal of Web Semantics*. 37–38 (2016) 184–206. doi:10.1016/j.websem.2016.03.003.

[24] A. Katasonov, and M. Palviainen, Towards ontology-driven development of applications for smart environments, in: 2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops), IEEE, 2010: pp. 696–701. doi:10.1109/PERCOMW.2010.5470523.

[25] J.Z. Pan, S. Staab, U. Aßmann, J. Ebert, and Y. Zhao, Ontology-Driven Software Development, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi:10.1007/978-3-642-31226-7.

[26] L. Vogt, Organizing Phenotypic Data—A Semantic Data Model for Anatomy, n.d.

[27] L. Vogt, R. Baum, C. Köhler, S. Meid, B. Quast, and P. Grobe, Using Semantic Programming for Developing a Web Content Management System for Semantic Phenotype Data, *Lecture Notes in Computer Science*. 11371 (2019).

[28] L. Vogt, R. Baum, P. Bhatty, C. Köhler, S. Meid, B. Quast, and P. Grobe, SOCCOMAS: a FAIR Web Content Management System that is based on Semantic Programming, n.d.

[29] GitHub: Source Code Ontology for Semantic Ontology-Controlled Web Content Management System (SOCCOMAS),

(n.d.).
https://github.com/SemanticProgramming/SOCCOMAS.

[30] T. Berners-Lee, Linked Data, (2009). https://www.w3.org/DesignIssues/LinkedData.html.

[31] Morph·D·Base: a morphological online data repository, (n.d.). https://www.morphdbase.de/.

[32] GitHub: Source Code Ontologies for semantic Morph·D·Base, (n.d.).
https://github.com/SemanticProgramming/SemMorphDBase.

[33] BioPortal, (n.d.). http://bioportal.bioontology.org/.

[34] Phenotype And Trait Ontology (PATO), (n.d.). http://obofoundry.org/ontology/pato.html.

[35] Semantic Morph·D·Base Prototype, (n.d.). https://proto.morphdbase.de.

[36] G. De Giacomo, and M. Lenzerini, TBox and ABox Reasoning in Expressive Description Logics, in: Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Morgan Kaufmann, 1996: pp. 316–327. doi:10.1.1.22.8293.