

MASTER'S THESIS

CONFIGURABLE SCHEMA-AWARE RDF DATA INPUT FORMS

DÁVID KONKOLY

MAY 2017



ALBERT-LUDWIGS UNIVERSITÄT FREIBURG
DEPARTMENT OF COMPUTER SCIENCE
CHAIR OF DATABASES AND INFORMATION SYSTEMS

Candidate

Dávid Konkoly

Matr. number

3757311

Working period

18. 10. 2016 – 16. 05. 2017

First Examiner

Prof. Dr. Georg Lausen

Second Examiner

Prof. Dr. Peter Fischer

Supervisor

Victor Anthony Arrascue Ayala

Abstract

Ontologies describing biomedical research processes are getting prevalent nowadays. Their utility is that they allow the replacement of textual documentation with structured data. This way the data can be exchanged among research institutes which supports knowledge sharing and collaboration. Nevertheless, the creation of RDF data for documentation purposes requires a software application offering some graphical user interface to achieve convenient data input. Usually, such applications are complex and require a large amount of code. Furthermore, the communication between domain experts and developers, and the specification and documentation of the software is often a tedious task. The goal of the thesis is to overcome this bottleneck of RDF data creation with a software framework which can be programmed in a high-level declarative way. With the implemented approach the development of data management applications can be significantly simplified and accelerated and made possible for researchers even without programming experience.

Kurzfassung

Für die Beschreibung von biomedizinischen Prozessen werden heutzutage immer häufiger Ontologien benutzt. Ihr Nutzen besteht darin, dass sie erlauben textbasierte Dokumentation durch strukturierte Daten zu ersetzen. Auf diese Art wird es Institutionen ermöglicht Daten auszutauschen, wodurch Kollaboration und die gemeinsame Benutzung von Wissen gefördert wird. Trotz allem erfordert die Erstellung von diesen resource description framework (RDF) Daten für Dokumentationszwecke eine Softwareanwendung mit benutzerfreundlicher Oberfläche, um bequeme Dateneingabe zu ermöglichen. Solche Anwendungen sind normalerweise komplex und benötigen eine große Menge Code. Darüber hinaus ist die Kommunikation zwischen Domainexperten und Softwareentwicklern und die Spezifikation sowie Dokumentation der Software oft eine langwierige Aufgabe. Das Ziel dieser Masterarbeit ist die Überwindung dieses Engpasses mithilfe eines Softwareframeworks, welches deklarative Programmierung auf hoher Ebene ermöglicht. Die Entwicklung von RDF Datenmanagement Anwendungen wird durch den im Rahmen dieser Arbeit implementierten Ansatz wesentlich vereinfacht und beschleunigt, sodass sie selbst Forschern ohne Programmiererfahrung ermöglicht wird.

Contents

| | |
|--|------------|
| Abstract | II |
| Kurzfassung | III |
| 1 Introduction | 1 |
| 1.1 RDFBones Project | 1 |
| 1.2 Goal of the thesis | 1 |
| 1.3 Thesis outline | 2 |
| 2 Preliminaries | 3 |
| 2.1 Semantic Web | 3 |
| 2.1.1 Resource Description Framework | 3 |
| 2.1.2 RDF and RDFS Vocabularies | 5 |
| 2.1.3 OWL | 7 |
| 2.1.4 SPARQL | 9 |
| 2.2 Ontologies | 12 |
| 2.2.1 Ontology for the anatomy of the human body | 12 |
| 2.2.2 Ontology for Biomedical Investigations (OBI) | 13 |
| 2.2.3 RDFBones ontology | 15 |
| 2.3 Web applications | 16 |
| 2.3.1 Fundamentals | 16 |
| 2.3.2 VIVO framework | 21 |
| 2.4 Related Work | 26 |

| | | |
|----------|--|-----------|
| 3 | Problem Statement | 27 |
| 3.1 | Multi level data input | 27 |
| 3.2 | RDFBones use-cases in VIVO | 30 |
| 3.3 | Solution scheme | 31 |
| 4 | Vocabulary for RDF data input | 33 |
| 4.1 | Elements of the vocabulary | 33 |
| 4.1.1 | Data definitions | 33 |
| 4.1.2 | Form definition | 35 |
| 4.2 | Use-cases of the <i>RDFBones</i> project | 36 |
| 4.2.1 | Primary Skeletal Inventory | 36 |
| 4.2.2 | Study Design Execution | 39 |
| 5 | Framework functionality | 43 |
| 5.1 | Main software modules and tasks | 43 |
| 5.1.1 | Validation | 44 |
| 5.1.2 | Dependencies and form functionality | 46 |
| 5.1.3 | Graph model generation | 48 |
| 5.2 | Implementation | 51 |
| 5.2.1 | Client side | 51 |
| 5.2.2 | Server side | 55 |
| 6 | Conclusion | 60 |
| 6.1 | Evaluation | 60 |
| 6.2 | Future work | 62 |

List of Figures

| | | |
|------|--|----|
| 2.1 | RDF graph notation | 4 |
| 2.2 | Class hierarchy of the RDF/RDFS vocabulary | 7 |
| 2.3 | A subset of OWL vocabulary | 8 |
| 2.4 | Scheme of the restrictions | 8 |
| 2.5 | Ontology extension | 9 |
| 2.6 | Example RDF dataset | 10 |
| 2.7 | Ontology structure for the human skeleton | 13 |
| 2.8 | Extending OBI ontology | 15 |
| 2.9 | RDFBones as OBI extension | 16 |
| 2.10 | Client server communication | 17 |
| 2.11 | Static and dynamic web pages | 20 |
| 2.12 | Faux properties on VIVO profile pages | 22 |
| 2.13 | Faux property descriptor RDF dataset | 22 |
| 2.14 | Custom entry form for skeletal subdivision | 23 |
| 3.1 | Ontology and RDF triples for complex entities | 28 |
| 3.2 | Multi level form | 28 |
| 3.3 | Skeletal subdivision graph pattern and form layout | 30 |
| 3.4 | Investigation graph pattern and form layout | 30 |
| 3.5 | Extended RDF graph pattern definition | 32 |
| 3.6 | Framework functionality outline | 32 |
| 4.1 | Java classes for RDF nodes | 33 |
| 4.2 | Java classes for RDF triples | 34 |

| | | |
|------|---|----|
| 4.3 | Java classes for the form | 35 |
| 4.4 | Java classes for instance viewer | 36 |
| 4.5 | Primary skeletal inventory extension | 37 |
| 4.6 | Skeletal inventory data triples | 37 |
| 4.7 | Generated interface for primary skeletal inventory | 38 |
| 4.8 | Complete data definition | 39 |
| 4.9 | UML object diagram for form layout definition | 39 |
| 4.10 | Glabella and its expressions | 40 |
| 4.11 | RDFBones extension for study design execution | 40 |
| 4.12 | Input form for study design execution | 41 |
| 4.13 | Input form for study design execution | 41 |
| 4.14 | Instance selector for existing bone segment | 42 |
| 4.15 | Complete data model | 42 |
| 4.16 | Extension schemes of the discussed use cases | 42 |
| 5.1 | More detailed scheme | 44 |
| 5.2 | Processor tasks | 44 |
| 5.3 | Valid and invalid nodes | 45 |
| 5.4 | Valid and invalid graph | 45 |
| 5.5 | Form element order | 46 |
| 5.6 | Skeletal inventory data constraints | 46 |
| 5.7 | Form dependency subgraphs | 47 |
| 5.8 | Form descriptor JSON | 49 |
| 5.9 | Conversion from triples into graph model | 49 |
| 5.10 | JSON/RDF through the graph model | 50 |
| 5.11 | Form and form element JavaScript classes | 52 |
| 5.12 | SubForm and sub form adder | 53 |
| 5.13 | UML class diagram for FormConfiguration | 56 |
| 5.14 | UML class diagram for the WebappConnector interface | 56 |
| 5.15 | Form loading process | 57 |
| 5.16 | UML class diagram for VariableDependency | 58 |
| 5.17 | UML class diagram for Graph | 59 |

Introduction

1.1 RDFBones Project

The master thesis is written in the frame of the project called *RDFBones*. The project is conducted by the Biological Anthropology Department of the Universitätsklinikum Freiburg and funded by the Deutsche Forschung Gemeinschaft (DFG). The main idea of the project is to make possible the definition of the rules and steps of particular anthropological investigations in a machine-readable way. This is achieved by the development of a core ontology in Ontology Web Language (OWL), that describes the general scheme of the processes, while custom problems are defined using so-called ontology extensions. The extensions are represented by further ontological RDF statements, which has the advantage that a web application for creating RDF data about the execution of the processes, can be programmed in a generic way so that it adapts its interface to the various cases [1]. During the project an open-source web application framework, called VIVO, is adopted and developed [2].

1.2 Goal of the thesis

Data creation about research processes happens through multiple different web pages, where each page is responsible for a particular subset of the data. The main structure of the individual input forms is characterized by the scheme of the data they create, which is in our case a subset of the core

ontology. While the elements of the pages, in turn, are defined in the certain extensions of the addressed scheme. The idea of the thesis is to develop a web application framework that is capable of generating its interfaces based on a declarative definition of the dataset they supposed to create. To achieve this, a descriptor language is designed which is capable of expressing RDF data models and their mapping to the interface. The utility of the idea is that the developed system can be applied not only for the concise solution of the problems of the *RDFBones* project but for any domain, where the rules are defined using ontology extensions.

1.3 Thesis outline

The second chapter conveys all the background information that is necessary to understand the problem solved by the thesis. The first two subsections handle the RDF data model and the ontologies applied in the project, while the third section is about the basics of the web application technologies and the VIVO framework. The third chapter contains the problem statement and explains briefly the scheme of the proposed solution. The fourth chapter presents the elements of designed descriptor language, and demonstrates how it can be applied to specify two use-cases of the *RDFBones* project. The fifth chapter is devoted to the discussion of the functionality of the developed software framework. Its first section provides an overview of the main components of the software, while the second gives a deeper insight into the implementation. The last, sixth chapter covers the conclusion and the evaluation of the achieved system and presents the further potential in the idea.

Chapter 2

Preliminaries

2.1 Semantic Web

2.1.1 Resource Description Framework

The Resource Description Framework (RDF) is a metadata data model used for representing information on the web. The data in RDF is organized into triples, where each triple consists of a subject, predicate, and object. The set of RDF triples constitutes a directed graph, which is referred as RDF graph. The nodes of the graph are the subjects and the objects, while the edges are the predicates [3].

The three most important types of node are the instances, classes and literals. An instance represents concrete an entity like a person, an institute, but can be an abstract concept. Classes are general concepts, to which the instances can belong, while literals represent data values assigned to instances. The following Listing shows some example triples that illustrates the basics of information representation using triples.

| | | | |
|------|------------|---------|---|
| Bob | instanceOf | Student | . |
| Math | instanceOf | Course | . |
| Bob | attends | Math | . |
| Bob | avgGrade | 1.73 | . |

Listing 2.1: Information in triples

In the example *Bob* and *Math* are instances, and they represent a concrete

person and a math course of some university. The *Course* and *Student* are classes and the value 1.73 is a literal. The values of the predicates are called properties. There are three main types of it, the one which connects instances to classes (*instanceOf*), one that expresses the relationship between instances (*attends*), and one which assigns a literal value to an instance (*avgGrade*). Important to mention that a literal value cannot be subjects of a triple.

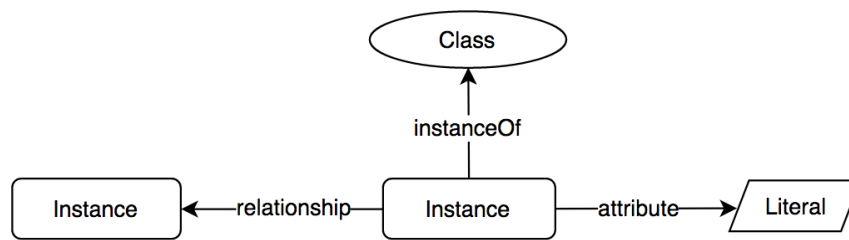


Figure 2.1: RDF graph notation

Figure 2.1 illustrates the three main types of node in RDF data. In the following this notation will be used in the figures, namely ellipse for the classes, rectangle for the instances and a rhombus for the literals.

The provided example gave just an insight into a triple based information representation, but the data in Listing 2.1 is not a valid RDF dataset. The instances and classes together are called resources (will be discussed later why), and each of them has to have an Internationalized Resource Identifier (IRI). The IRIs have to be unique, thus if someone wants to represent information in RDF, then it is necessary to choose an own namespace for own IRIs. For example if the chosen namespace is `<http://myDomain.com#>`, then the IRI of the class *Student* may be `<http://myDomain.com#Student>`. The IRIs of the instances in most of the cases are not given manually but generated by the software, and their names like *Bob* or *Math* are stored in literal values assigned to them.

RDF data does not contain only IRIs from own namespaces, but there are vocabularies that offer a set a built in IRIs. The three most important vocabularies are the RDF, RDF Schema (RDFS), XML Schema (XMLS). RDF and RDFS offer classes and properties, while XMLS contains the datatype IRIs for literal values. Each them of has its namespace, which is for the sake of readability often abbreviated with prefixes. In such cases, the IRIs are

represented by a "prefix:suffix" syntax, which means the concatenation of the prefix and the suffix. The following Listing shows a valid RDF dataset containing the triples of the example.

```
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
@prefix xmls:   <http://www.w3.org/2001/XMLSchema#>
@prefix domain: <http://myDomain.com/#>

domain:1      rdf:type          domain:Student    ;
               rdfs:label       "Bob"^^xmls:string .
domain:2      rdf:type          domain:Course     ;
               rdfs:label       "Math"^^xmls:string .
domain:1      domain:attends    domain:2         ;
               domain:avgGrade  "1.72"^^xmls:float .
```

Listing 2.2: RDF data in N3 serialization format

The ";" in N3 is used to divide predicate-object pairs that constitute a triple with the same subject. With this approach, the subject does not have to be written as many times as many triples it participates in. The property *rdf:type* expresses the *instanceOf* relationship, while *rdfs:label* is the most widely used property to assign names to instances. By the literals, it can be seen that they are surrounded by quotation marks and extended with the type notation from the XMLS vocabulary.

2.1.2 RDF and RDFS Vocabularies

In the previous section, the RDF and RDFS vocabularies were already mentioned, but this section provides much more detailed information about their usage and utility. RDF and RDFS offer classes and properties that allow expressing the rules of the RDF data [4]. First of all, it has to be defined, what IRIs represent classes and what properties. For this purpose there are two classes the *rdfs:Class* and *rdf:Property*. To define own properties and classes, just like by the instances, the *rdf:type* property have to used.

```
domain:Student      rdf:type          rdfs:Class .
```

| | | |
|----------------|-----------|---------------|
| domain: Course | rdf: type | rdfs: Class . |
|----------------|-----------|---------------|

Listing 2.3: Class definition

In this way, it is defined that in our dataset the instances will represent students and courses. The definition in the case of the properties is a bit more complex because it has to be defined too, what type of subjects and objects they can connect. For this definition there are two properties in RDFS, the *rdfs:domain* and the *rdfs:range* respectively.

| | | |
|------------------|--------------|-------------------|
| domain: attends | rdf: type | rdf: Property ; |
| | rdfs: domain | domain: Student ; |
| | rdfs: range | domain: Course . |
| domain: avgGrade | rdf: type | rdf: Property . |
| | rdfs: domain | domain: Student . |
| | rdfs: range | xmls: float . |

Listing 2.4: Property definition I.

The further really important concept in data modeling is the subclass relationship. This is expressed in RDFS with the the *rdfs:subClassOf* property. The informal definition of the subclass concept is that if class B is a subclass of class A, then every instance of class B is an instance of class A as well. The definition is that every student is a person can be done in the following way:

| | | |
|-----------------|------------------|------------------|
| domain: Person | rdf: type | rdfs: Class . |
| domain: Student | rdfs: subClassOf | domain: Person . |

Listing 2.5: Sub class definition

The previous three Listing contained the RDF triples for the definition of an own vocabulary. The vocabulary is called as well ontology and its purpose is to define the scheme of the data. Nevertheless, the properties for label, type, domain and range, and the classes for class and property are supposed to be defined like the ones in the example ontology. The RDF and RDFS vocabulary have this definition too, and it is depicted in Figure 2.2.

The root of any RDF dataset is the class *rdfs:Resource*. *rdfs:Resource* is class because it is the instance of the *rdfs:Class*. *rdfs:Class* is a class

with the given classes and property. In that case, the second class is assigned to the restriction node with the *owl:onClass* property.

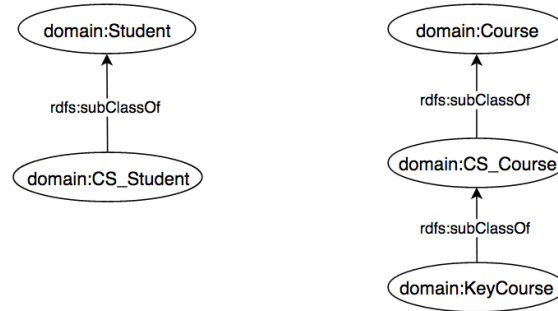


Figure 2.5: Ontology extension

To illustrate the utility of restrictions, the ontology from the previous section was extended with three subclasses (Figure 2.5). The abbreviations *CS* stands for computer science. The following two restriction asserts that a computer science student is allowed to attend only computer science courses, and has to take at least two key courses.

```

domain:CS_Student rdfs:subClassOf [
  rdf:type          owl:Restriction ;
  owl:onProperty   domain:attends ;
  owl:allValuesFrom domain:CS_Course .
]

domain:CS_Student rdfs:subClassOf [
  rdf:type          owl:Restriction ;
  owl:onProperty   domain:attends ;
  owl:onClass      domain:KeyCourse;
  owl:minQualifiedCardinality "2"^^xmls:nonnegativeInteger .
]

```

Listing 2.7: Blank nodes in N3

2.1.4 SPARQL

RDF data is stored most commonly in triplestores. A triplestore is a software for storage and retrieval of RDF triples. The retrieval, like by any other kind

of database, happens via queries, which are formulated in a particular query language. The query language for RDF is called SPARQL.

A basic SPARQL query consists of two main parts. Firstly of a triple pattern, which differs only from an RDF dataset that it contains variables. The variables are denoted with a question mark in the triple pattern. Secondly of a set of variables that the query has to return. Listing 2.8 shows an example query to demonstrate the syntax. After the *SELECT* keyword there is the variable to return and after the *WHERE* keyword inside the parenthesis, there is the triple pattern.

```
SELECT ?student
WHERE {
  ?student    rdf:type    domain:Student .
}
```

Listing 2.8: SPARQL Query I.

It can be seen that the namespace abbreviation with prefixes works the same way like in RDF documents. If the query is executed on the data set from Figure 2.6, then it returns the *domain:1* and *domain:3* instances for the variable *?student*.

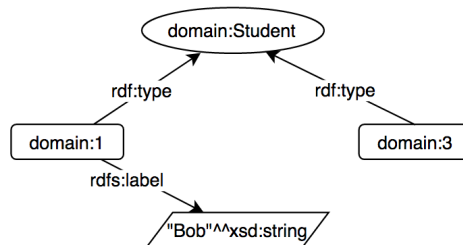


Figure 2.6: Example RDF dataset

Furthermore, the triple pattern can consist of multiple triples as well, like in the following query:

```
SELECT ?student ?label
WHERE {
  ?student    rdf:type    domain:Student .
  ?student    rdfs:label  ?label .
}
```

```
}

```

Listing 2.9: SPARQL Query II.

Executed on the same dataset, this query returns only the instance *domain:1*, because *domain:3* does not have a label. Thus there is no matching RDF node for variable *?label*. This means that the triples in a triple pattern are in *AND* relationship. But there is the keyword *OPTIONAL*, which allows the definition of sub triple patterns, which are not required in the RDF dataset. So if the triple with the label variable is in the *OPTIONAL* subpattern, then the query II. returns *domain:3* instance as well, so that the variable for the label will be empty.

```
OPTIONAL { ?student rdfs:label ?label . }
```

Listing 2.10: Optional sub triple pattern.

Moreover, it is possible to define filters on the variables. The most commonly used are the regular expression filter on literals.

```
FILTER regex(?label , "^Bo")
```

Listing 2.11: Regex filter in SPARQL

Finally, it is possible in SPARQL to query blank nodes, the same way with variables. The following query then queries the ontology and returns at least how many key courses must be attended by a computer science student.

```
SELECT ?minKeyCourses
WHERE {
  domain:CS_Student rdfs:subClassOf ?restriction .
  ?restriction      rdf:type          owl:Restriction .
  ?restriction      owl:onProperty  domain:attends .
  ?restriction      owl:onClass     domain:KeyCourse .
  ?restriction      owl:minQualifiedCardinality ?minKeyCourses .
}
```

Listing 2.12: SPARQL Query III.

2.2 Ontologies

Ontologies are used to describe types, relationships, and properties of objects of a certain domain. It is a common practice to use existing ontologies rather than developing them by ourselves. The main reason for this lies in the fact that the ontology development is a time consuming and tedious process. Two ontologies are taken in the project, one for the human anatomy and one for biomedical investigations. To connect these two, during the project our ontology (called as well *RDFBones*) is developed too. Firstly the two applied ontologies will be discussed and lastly the *RDFBones* ontology.

2.2.1 Ontology for the anatomy of the human body

The ontology modeling the human body is called *Foundational Model of Anatomy* (FMA). FMA is a fundamental knowledge source for all biomedical domains, and it provides a declarative definition of concepts and relationships of the human body for knowledge-based applications. It contains more than 70 000 classes, and 168 different relationships [6]. All kind of anatomical entities is represented in FMA, like molecules, cells, tissues, muscles, and of course bones. In our project, we use only the skeletal system related subset of the FMA. The taken elements are the following classes (and its subclasses) and properties:

- Classes

Subdivision of skeletal system - fma:85544

Bone Organ – fma:5018

- Properties

fma:systemic_part_of

fma:regional_part_of

fma:constitutional_part_of

The class *Bone Organ* is the superclass of all bones in the human skeleton. Each bone belongs to a subclass of the class *Subdivision of skeletal system*. Moreover, there are such skeletal subdivisions which are part of another skeletal subdivision. In both cases, the relationship is expressed by

the property *fma:systemic_part_of*. To define which bone organ belongs to which skeletal subdivision, FMA contains *owl:someValuesFrom* restrictions (Listing 2.13).

```
fma:BoneOrganX  rdfs:subClassOf [
  rdf:type          owl:Restriction ;
  owl:onProperty   fma:systemic_part_of ;
  owl:someValuesFrom fma:SkeletalSubdivisionY .
]
```

Listing 2.13: Rules of the skeletal system defined in OWL

These restrictions mean that a bone organ instance cannot stand on its own, but it has to be a systemic part of an appropriate skeletal subdivision instance. Figure 2.7 shows the main structure of the applied subset of FMA by depicting the restrictions with red arrows, and the subclass relationships with dashed arrows.

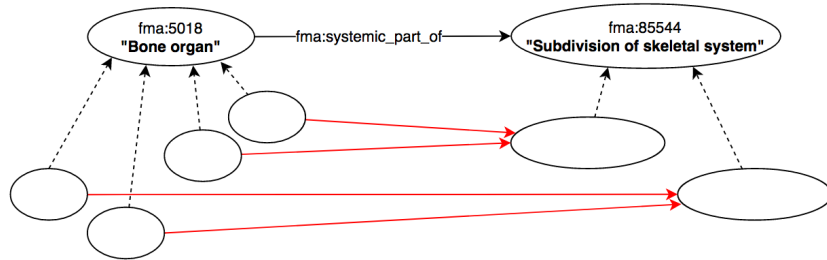


Figure 2.7: Ontology structure for the human skeleton

Finally, the advantage of using the FMA ontology is that, if in the future further elements of the human body have to be addressed by the research processes, i.e. muscles, then these classes can be easily integrated to the currently applied subset.

2.2.2 Ontology for Biomedical Investigations (OBI)

The aim of the OBI ontology is to provide a formal representation of the biomedical investigations in order to standardize the processes among different research communities. It is a result of a collaborative effort of several working groups, and it is continuously evolving as new research methods are

being developed. Its main function is to provide a vocabulary that allows the definition of the rules regarding how biological and medical investigations have to be performed. OBI reuses terms from the *Basic Formal Ontology* (bfo), from the *Information Artifact Ontology* (iao) and from the *Open Biological and Biomedical Ontologies* (obo) [7]. The most important classes and properties adopted by the *RDFBones* project are the following ones:

- Classes

Investigation - obo:0000015

Process - bfo:0000015

Entity - bfo:0000001

Information Content Entity - iao:0000030

- Properties

has part - bfo:00000051

has specified input - obi:00000293

has specified output - obi:00000299

Above these classes several other classes have been taken, but they are sufficient to discuss the essence of the ontological definition of investigations. The idea of *RDFBones* project is to define custom investigations by defining subclasses of the above-mentioned classes, and restriction on the properties, the same way like FMA describes the human skeleton. These further ontological statements are called extensions. The following image illustrates an example ontology extension for an investigation (the notation is the same like on Figure 2.7, just the restrictions are defined on the OBI properties).

It can be seen that the investigation contains two processes (*has part* predicate), and each process has various inputs and output. In our case, input entities are segments of bone organs but they could be any material. During a particular process the input entities are studied, and the result of the study is an information content entity instance, which represents some measurement value. The provided scheme is just an illustrative example and the modeling of an investigation in reality is more complex and involves more OBI classes. They will be discussed in a bit more detail in Chapter 4.

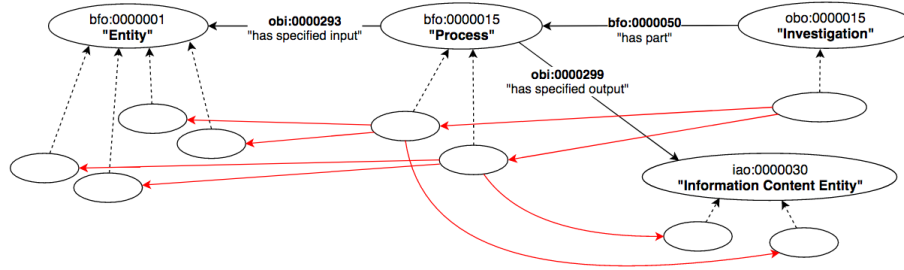


Figure 2.8: Extending OBI ontology

The conclusion is that with these classes and properties, OBI offers a powerful vocabulary for defining custom investigations. The advantage of such ontological description is that everything is stored using RDF triples, and therefore it is possible to develop software applications which generates user interfaces based on the results of SPARQL queries performed on the extensions. From this follows that these formal definitions can be considered as software specification as well.

2.2.3 RDFS ontology

RDFS ontology is an extension of the OBI ontology. It has its own namespace (the empty string a valid prefix too):

```
PREFIX : <http://w3id.org/rdfbones/core#>
```

Listing 2.14: RDFS ontology namespace

Figure 2.9 shows the four most important classes (blue ones) as subclasses of further OBI classes using the above-defined prefix. On the left, there is class *:SkeletalInventory*, whose purpose is to incorporate the set of existing bones in a skeletal collection. The *Completeness2States* and *Completeness2StatesLabel* classes are used to represent if a bone segment is complete or just partly present. The *:SegmentOfSkeletalElement* is the subclass of *bfo:0000020*, which is the subclass of the previously mentioned *Entity* class. With this class for bone segments, *RDFS* makes possible the definition of custom segments of particular bone organs, in order to allow more fine-grained modeling of the research activity. It is again a sort of ontology

extension, thus the definition has to be done by means of custom subclasses and restrictions. The property that expresses the relationship between bone organs and bone segments is the *fma:regional_part_of*. Also this is the property that connects the RDFSkeletons and thus the OBI to the FMA.

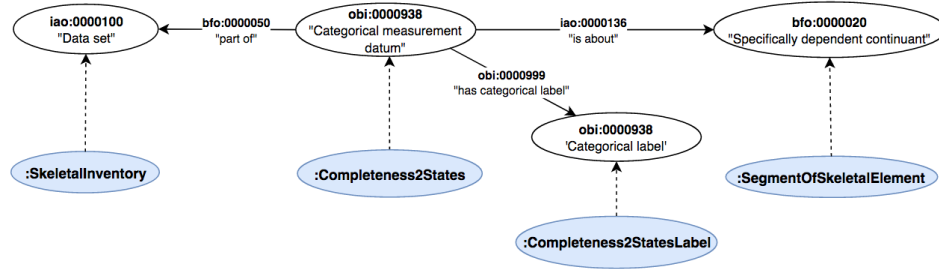


Figure 2.9: RDFSkeletons as OBI extension

2.3 Web applications

Nowadays web applications are prevalent since they have the advantage against a desktop application that they do not have to be installed on a local computer, and can be accessed from the web browser and thus can be used anywhere from the world. Like by the ontologies, the development of a web application is a complex process, and therefore an existing open-source framework called VIVO has been applied in the project. VIVO is an appropriate choice because it has been developed particularly for browsing and editing RDF data. The designed software of the thesis is an extension of the VIVO framework. To provide the necessary information required to understand the problem and the solution of the thesis, this section covers some fundamental web application technologies and the main functionality and capabilities of VIVO framework.

2.3.1 Fundamentals

Client-server architecture

Web applications consist of two main units. The first is the client, which the user interacts with, and the second is the server, which is the other application that serves the request coming from the client. The client and server

programs are running normally on different machines, and they communicate through Hypertext Transfer Protocol (HTTP). The main mechanism is that client, which is a web browser application, sends an HTTP request through the web, and the server is found based on the URI of the HTTP request. Upon the content of the request, the server returns a document written in Hypertext Markup Language (HTML). The HTML document contains the definition of the elements of the interface and it is interpreted by the web browser.

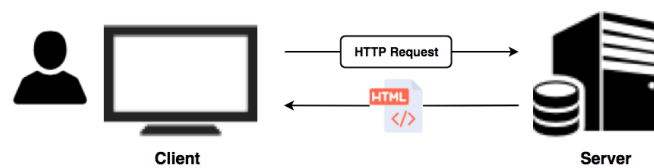


Figure 2.10: Client server communication

An HTML document consists of different elements like buttons, tables input fields. Each element is represented by so-called tags. The HTML has a hierarchical structure, and each element is the child of the tag *html*. On listing 2.15 it can be seen that each tag has an opening and closing element.

```
<html>
  <div class = "welcome"> Welcome on the web application </div>
  <a href="http://webapp.com/page1"> Page 1 </a>
</html>
```

Listing 2.15: Example HTML document

The tag, like the *html* in the example can contain further tags, and have normally at least one parameter (i.e *class* and *href*). The tag *div* is the most general element of web pages, while the *a* tag defines a link, where the *href* parameter defines the URI of the HTTP request they initiate. The request of the example link arrives at the same server, and the task of the application is to process the request URI and return the HTML document for the new page.

Data driven web applications

Most of the web applications nowadays incorporate databases. Databases are used to store large amount data in an organized way. Databases are always come along with a database management system (DBMS) that allows creating, edit, delete and retrieve data in the database. So DBMS is software that acts as an interface between the web application and the data. In the following, the database and DBMS together will be referred simply as database.

By web applications using databases the most important point, that web pages are not statically defined in HTML files, but generated by the application dynamically using a particular dataset. The process of loading of web pages showing any data starts with the execution of a query. This means the web application sends a query to the database and gets the desired data. The result of a query regarding the web application is conventionally a list of data objects. The term object is generally used and refers to a data type that organizes its values by keys. The elements of the output list represent the rows of the query result table, and the fields of the object, in turn, the rows respectively.

To define how the web page has to be generated from the dataset a so-called template file are used. The template file is an HTML document, that is extended with some additional syntax. The elements defined in this syntax can be interpreted by the template engine. There a lot of template engines and languages, but in the following, I provide an illustrative example in *Freemarker* template language, that is used within the VIVO framework.

```
<#list students as student>
  <tr>
    <td>${student.name}</td>
    <td> <@linkButton "grades" ${student.id} /> </td>
  </tr>
</#list>
```

Listing 2.16: Template file example

Important that the data that is passed to the template engine has a name, by which it is referred to the template file. In the example query result is

stored in a variable *students*, where each student object has two keys, the *name* and *id*. The tag *#list* represent a loop, which iterates through the input list. The content withing the *#list* tag appear as many times in the resulting HTML, as many elements the input list have. The variables withing normal HTML tags are accessed with *\${..}*.

Other useful feature of templates that it allows the definition of the macro, which acts like subroutines in programming. In the example the macro *linkButton* takes two input parameter and generates the *<a/>* tag with a certain image. This makes the development more convenient and clear.

```
<#macro linkButton urlMapping id>
  <a href="webapp.com/${urlMapping}?id=${id}">
    
  </a>
</#macro>
```

Listing 2.17: Macro definition

In the macro definition, it can be seen that the URL of the link contains the parameter *id*, which is additional information in the HTTP request. The idea is that the request is handled by such a server routine that substitutes the value of the parameter into a query, to get data about individuals. Then the returned page may contain an additional link for further data entries. This is the fundamental method how web pages are used to discover data from databases.

Interactive web pages

The previous section showed the principles of how data can be browsed using web pages and links. In such static cases, the HTML document was assembled completely by the server, and the links initiated the loading of whole new pages. Nevertheless, it is often more efficient and leads to better user experience if the new content is added dynamically to the currently opened web page. Such functionality can be achieved with JavaScript (JS), which is a scripting language run by the web browser. The most fundamental features of JS that it is capable of storing data in variables and can add, edit

or remove HTML elements of the pages.

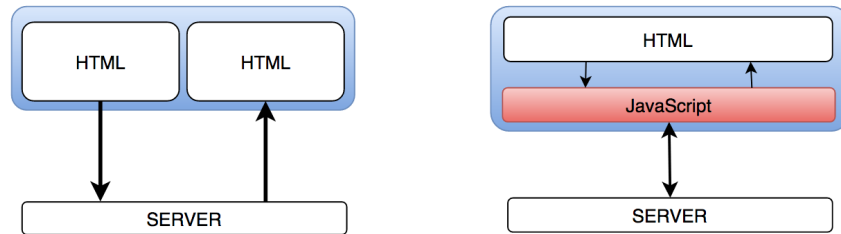


Figure 2.11: Static and dynamic web pages

Figure 2.11 illustrates the difference between static and dynamic web pages, where the blue rectangle stands for the client side. The idea of dynamic web pages in data driven web applications, which if new content has to be shown on the page, then not an HTTP request is sent to the server, but a JS routine is called. JS code is defined in the HTML document within the `<script/>` tag. The routines are assigned to HTML elements by their ids in the following way:

```
<html>
  <div id = "button"> Show content </div>
  <div id = "content"></div>
  <script>
    ...
    $( "#button" ).click( function() {
      $( "#content" ).text( data )
    })
    ...
  </script>
</html>
```

Listing 2.18: JavaScript routine assigned to an HTML element

This simple JS code illustrates how it is possible to load new content to the HTML element. In the example the *append* function sets the text of the div with id *button*. The *data* is a JS variable holding some text. The value variable can be either initialized by the server by the page assembly, or by AJAX calls. AJAX is an acronym for *Asynchronous JavaScript and XML*. The AJAX is technology that allows JS to load data from the server

asynchronously, which means that the request is initiated through JS routine, and response arrives as well to JS routine. The data by AJAX is mostly in JSON format. JSON stands for *JavaScript Object Notation*, which is a standard data format. A JSON object consists of a set of key-value pairs, where the value can be any data, arrays or even further objects.

```
{ key1 : "data",  
  key2 : [ "value1", "value2" ],  
  key3 : { key : "value" } }
```

Listing 2.19: JSON object example

2.3.2 VIVO framework

VIVO maintains two basic types of page. The first is for displaying, and the second is for creating and editing RDF data. The former will be referred to as profile page because it shows the data related to one individual RDF instance, and the latter will be referred to as an entry form. Although the scope of the thesis is only the data input, the profile pages will be covered as well briefly in the first part, because the entry forms are dependent on them to some extent and they are relevant regarding the future work. The second part provides a simplified explanation of how the RDF data input works originally in VIVO, which is necessary to understand the utility of the idea proposed by the thesis.

Profile pages

The task of the profile pages in VIVO is to display all instances and literals that constitute a triple with the instance, whose page has been called. These neighbor RDF nodes are shown on the pages grouped by property, and properties are organized into tabs. Further grouping possibility is offered by the so-called faux properties. Using faux property definition, it can be achieved that the instances on the profile pages are grouped by their type.

Figure 2.12 shows the layout of a VIVO profile page of a skeletal inventory, where in the skeletal subdivision property group the two faux properties (*skull* and *vertebral column*) can be seen. The instances (blue



Figure 2.12: Faux properties on VIVO profile pages

entries) are both connected to the skeletal inventory instance with the *rdfbones:hasSkeletalSubDivision* property, but as there two faux properties are defined for the classes "*fma:5018 -Skull-*" and for "*fma:13478 -Vertebral Column-*", they appear in distinct property fields. The peculiarity of VIVO that these interface related issues are not defined in template files of server routines, but in RDF configuration triples. Figure 2.13 a simplified dataset of the faux property description for the skull. The notation in the rectangle denotes the label of the instance because its IRI is not relevant at this point. By the profile page generation, only the triples regarding range class and the base property are considered, the value of the custom entry form is for the data input.

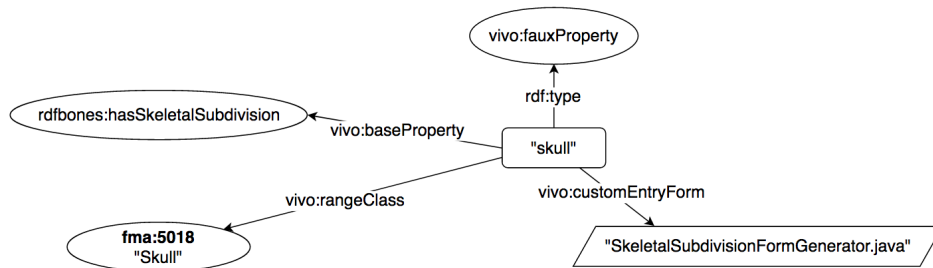


Figure 2.13: Faux property descriptor RDF dataset

On Figure 2.12 next to the property field, a button with a + sign can be seen, which is a link to a data entry form page. From each property field, the request arrives at the same handler routine on the server. These requests always have three parameters (VIVO uses in its parameter names URI instead of IRI, which is an abbreviation for Unified Resource Identifier,

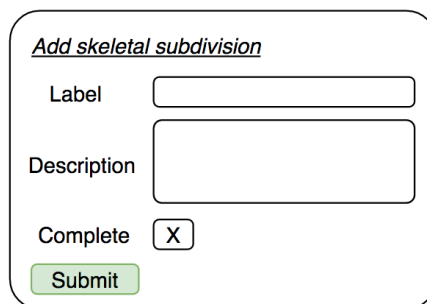
but it does not make a difference):

- *subjectUri* - IRI of the instance to whom the profile page belongs
- *predicateUri* - is the IRI of the property (can be a faux property)
- *rangeUri* - IRI of the range class of the property

The different entry forms can be called for editing the added data as well. This is done by the link (pen image) next to the blue data entries. VIVO knows that this is the case because the HTTP request contains an additional parameter called *objectUri*, which contains the IRI of the particular entry. In the edit case, the same entry form appears, and its fields are filled with the previously created data.

Custom entry forms

By default, VIVO redirects the user to such an entry form where only one instance or literal can be added with the type defined in the *rangeUri* parameter. However, it is possible to so-called entry custom forms for more complex datasets.



The image shows a web form titled "Add skeletal subdivision". It contains three input fields: "Label" (a single-line text box), "Description" (a multi-line text area), and "Complete" (a checkbox with an 'X' inside). Below these fields is a green "Submit" button.

Figure 2.14: Custom entry form for skeletal subdivision

Figure 2.14 shows the layout of an HTML form with three fields, where the user can enter specific literal values for a new skeletal subdivision instance. In an HTML form, each field has a variable name, based on the entered value can be identified by the server. By clicking the submit button, an HTTP request is sent to the server which contains the data in key-value pairs.

The idea of VIVO, that it allows the definition only of the HTML input form layout and the dataset, and the developer does not have to care about how the values from the form are substituted into the RDF data and how the new instances are generated. For the definition two files have to be created, a *Freemarker* template file (with .ftl extension) for the form definition and a Java class that contains in its fields the reference to the template file and the RDF data definitions. The name of the Java class is stored in the faux property definition (Figure 2.13), and thus VIVO can find it based on the *predicateUri* parameter of the custom entry form HTTP request. The template file definition for the form is simple due to the *Freemarker* macros provided in VIVO. Each type of input field has a distinct macro and have two input parameters, the parameter name, and the title. The macros do not only show the appropriate input fields but by the edit mode, they care about that the fields show the existing values.

```
<form>
  <@title "Add skeletal subdivision">
  <@labelField "label" "Label">
  <@textField "desc" "Description">
  <@booleanField "complete" "Complete">
  <@submitButton>
</form>
```

Listing 2.20: subdivision.ftl

The Java class that contains the necessary data has to extends the class *EditConfigurationGenerator* class, therefore its fields are given. The variable *templateFile* must hold the name of the template file.

```
this.templateFile = "subdivision.ftl"
```

Listing 2.21: Form definition in Java

The next fundamental variable of the generator class is the *triplesToCreate*, which contains the RDF triples to be created after the submission.

```
this.triplesToCreate = "
  ?subjectUri    rdfbones:hasSkeletalSubdivision    ?subDivision.
```

```

?subDivision    rdf:type          ?rangeUri .
?subDivision    rdfs:label        ?label^^xmls:string.
?subDivision    domain:description ?desc^^xmls:string .
?subDivision    domain:complete   ?complete^^xmls:boolean . "

```

Listing 2.22: RDF Triples to create

It can be seen that there are variables like in SPARQL, defined with a question mark. These must be the same as the variable defined in the template for the input fields. Moreover, there are three basic types of variables defined by the following three lists:

```

this.urisOnForm = {};
this.literalsOnForm = {"label", "desc", "complete"};
this.newInstances = {"subDivision"};

```

Listing 2.23: Variable type definition

There are no URIs (IRIs) on the form, but there are three literals. The variable *subDivision* has to get new unused IRI, and the *subjectUri* and *rangeUri* values are coming from the initial request. Moreover important part of the definition is the SPARQL query that retrieves the values for the form variables if the form is called for editing the existing data. For this purpose VIVO maintains a variable with the *Map<String, String>* with the name *sparqlForLiterals*. In this field the keys are variable names and the values are the SPARQL queries. By the form data loading for editing, VIVO executes the defined queries and passes the values in the variables to the template engine for the FTL macros.

```

this.sparqlForLiterals.put("desc", "
SELECT ?desc
WHERE {
    ?subDivision    domain:description    ?desc .
}")

```

Listing 2.24: SPARQL forExisting variable definition

This section gave an insight into how it is possible to define data input processes in a declarative way within the VIVO framework.

2.4 Related Work

The two most closely related Semantic Web based application for data acquisition are described in [8, 9]. Both systems incorporate domain ontologies in OWL and a certain mapping of them to the input form producing RDF data. The aim of the first application is to gather historical data for museums, and its forms consist solely of literal fields. It uses the CIDOC Conceptual Reference Model [10] to establish the connection with data input forms and the RDF data path. Furthermore, OWL restriction of the domain ontology is considered to decide if a field is mandatory or not. The scope of second application is a generic web survey system for clinical functional assessment. The content of the entry forms, namely the questions, subquestions and possible answers are defined in OWL ontologies. For the form layout, a *datamodel* ontology is designed, that addresses the domain ontology elements, and it allows the creation of configuration data, based on which the framework generates the necessary HTML and JS code. The source-code is available on GitHub. However, the documentation neither for the software and nor for the domain ontologies is enough detailed to make possible the adoption of their implementation in our project.

[11, 12] are discussing the main features of the declarative programming language, called Curry. Curry offers a high-level abstraction from application programming. Thus it helps the rapid construction of the interfaces. With Curry not only the data input forms can be developed, but any general purpose application, such as games. Both functional and logic programming features are supported by Curry, and it offers library for type oriented construction of the web user interfaces. It allows the definition of variables and custom routines and event handler functions. So it offers a lower level abstraction then the previous two systems and the thesis. Curry offers libraries as well for the server side, for database access [13], but it is only applicable to relational data model.

Chapter 3

Problem Statement

The first section discusses the problem in a general way by addressing the challenges of the implementation both for the client and the server side in case of more elaborate data input processes. The second section, in turn, refers to the use-cases of the *RDFBones* project and to the limitations of the VIVO framework. Finally, the third part introduces the main elements of descriptor logic and gives an insight into the functionality of the developed system.

3.1 Multi level data input

In the previous chapter it has been discussed how can we implement simple data input forms within the VIVO framework. The simplicity of the illustrated problem lied in that the number of instances which were created through the process was constant, in particular, one, and only a set of literal attributes were set by the user through HTML input elements. Nevertheless, there are more complex entities consisting of several subparts, where these subparts are represented in the ontology with further classes. Consequently, the RDF dataset for such entities incorporates multiple instances organized into a tree structure. Figure 3.1 shows an example ontology and an RDF dataset. The classes (ellipses) without notation are subclasses of the three main classes and their names are not relevant.

Such dataset poses the requirement for the input form, that the user has to be offered such interface elements which enable to add the components



Figure 3.1: Ontology and RDF triples for complex entities

and subcomponents step by step. Adding a component means in terms of the form, that a sub form has to appear which contains further input fields for the component instances.

Figure 3.2: Multi level form

Figure 3.2 shows the layout of the form for multi-level data. The additional element compared to the static HTML form is the field with a button for adding the sub forms. The dotted rectangle stands for the element which encompasses the added sub forms. The form data contains the same way the key-value pairs for the main form element, but it has an additional key, where the value is an array of the data objects of the sub forms. The sub

form data object works the same way, if it has sub forms then it contains further arrays. To realize such functionality, JavaScript routine is required on the form, that adds the sub form elements to the container, and generates the appropriate form data upon the user actions. Listing 3.1 shows the JSON object generated by the form from Figure 3.2, where the objects are surrounded with ("{}"), while the arrays with ("[]"). After the submission, the server has to process this object by iterating through the arrays of them and generate the appropriate RDF triples.

```
{ type : "eq:elementA",  
  label : "Element_4391",  
  components : [  
    { type : "eq:componentA1",  
      label : "Component_8531",  
      subComponents : [ { ... } ],  
    }, {  
      ...  
    } ]  
}
```

Listing 3.1: Multi level form data in JSON

A further challenge for the client algorithm is that the options of the selectors for the component type on the subforms are dependent on the selected type on the parent form (the form which the subform has been added from). This means that the client has to load asynchronously the values, by sending an AJAX request containing the selected type value to the server. The task of the server is to perform the query that retrieves the classes defined through restrictions in the ontology and to return the results. The aim of this functionality is first to ensure that only such data is created that conforms to the rules defined in the ontology, secondly, the interface is much more usable if not all the component classes are listed, just the ones that belong to the selected element class. Moreover, in this way, the validation on the server side after the submission can be omitted.

Finally, really important part of the problem is the editing of the existing data. By editing, the application has to restore the state of the form, in which it has been initially submitted. But since the form data is in this case, not

just a set of key-value-pairs but a multi-level data object, an algorithm is required that generates the multi-level JSON object from the existing triples iteratively. Then another routine on the client has to reset the state of the form with the appropriate subforms and certain selector options as well based on the arrived data. Moreover, both the client and the server has to be able to handle deletion of the particular subforms, which is done through AJAX.

3.2 RDFS use-cases in VIVO

In VIVO the data input problems were solved using static HTML forms written with *FTL* macros, and defining the graph pattern and variable types by assigning values of the fields of a certain Java class, based on which a generic algorithm creates and retrieves the data. Figure 3.3 and Figure 3.4 illustrate the graph pattern and the multi-level form layout of the two most important cases of the *RDFS* project.

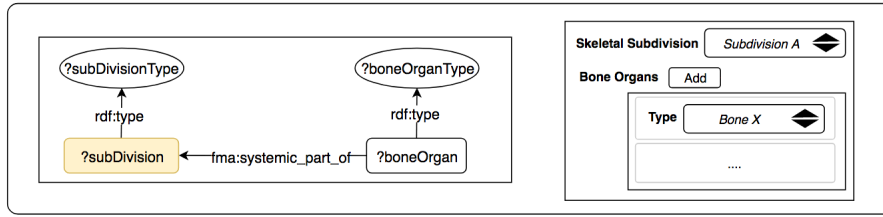


Figure 3.3: Skeletal subdivision graph pattern and form layout

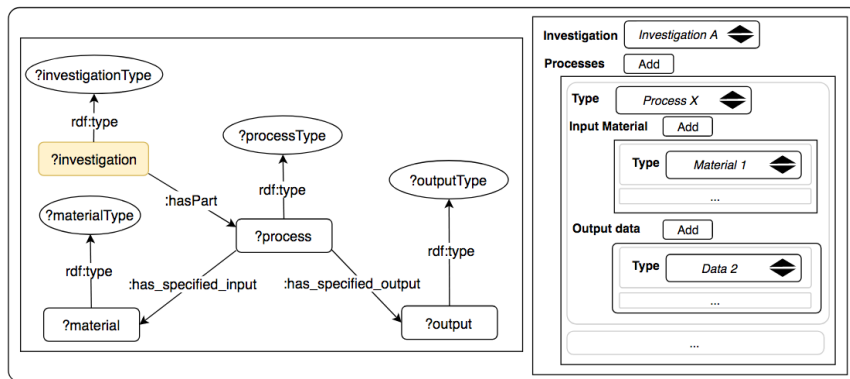


Figure 3.4: Investigation graph pattern and form layout

The problem is that it would be necessary to define custom JavaScript routines for each case, which adds the subforms and handles the dependencies between the particular selector fields because the FTL library is not able to designed to handle the dynamic events. Moreover, since VIVO cannot process the multi-level JSON object coming from the client based on a single, triple pattern in a string, for each case an individual Java routines have to be written that creates the data, as well as for the retrieval. Furthermore, the individual variables in the triple pattern do not appear only once in the result dataset. Thus their value cannot be defined with a single SPARQL query for the data retrieval.

3.3 Solution scheme

The idea of the thesis is to allow the declarative definition of the multi-level data input cases, in such a way that VIVO allows it for the static ones so that the various problems could be solved without writing individual Java and JavaScript routines. This is achieved by an additional set of descriptor Java classes and routines both on the client and the server side, which are capable of interpreting the extended descriptor objects.

Important difference wrt. VIVO that the scheme of the forms is defined through Java objects not in FTL files. The two main classes are the *Form* and the *FormElement*. There are subclasses of the *FormElement* class, which represent the different types of an input field. The form element that allows the definition of the multi-level form is the *SubformAdder* class. From the Java objects describing the multi-level layout, a JSON object is generated, which is interpreted by a JavaScript library, which generates the form and manages the functionality.

In the data definition, the most fundamental improvement is that the triple pattern is not expressed as a string but as a set of different types of the triple. The three main types of triple are the *Triple*, the *MultiTriple* and the *RestrictionTriple*. The *Triple* has the same role as a triple substring in VIVO, while the *MultiTriple* allows the expression of the hierarchy in the graph pattern, which presents on the form as well. With the *MultiTriple* the submission handler routine is prepared that a set of variables do not appear as single values in the form data, but within further objects in an

array. Likewise, in the other direction by the data retrieval the *MultiTriple* is the basis of the multi-level form data JSON object generation. The *RestrictionTriple* is used to express dependencies between classes in the graph pattern, which is relevant for the client. Figure 3.5 depicts the extended graph pattern definition for the general case, where the restriction triple are depicted with the red arrow and the multi triple with the double line arrow.



Figure 3.5: Extended RDF graph pattern definition

Above the data generation based on the graph pattern, an important utility of the Java library on the server the is capable of generating the SPARQL queries both for the data retrieval and for the data dependencies on the form, thus allowing more compact definition. Finally, an image is provided to illustrate the main elements and mechanisms of the implemented framework.

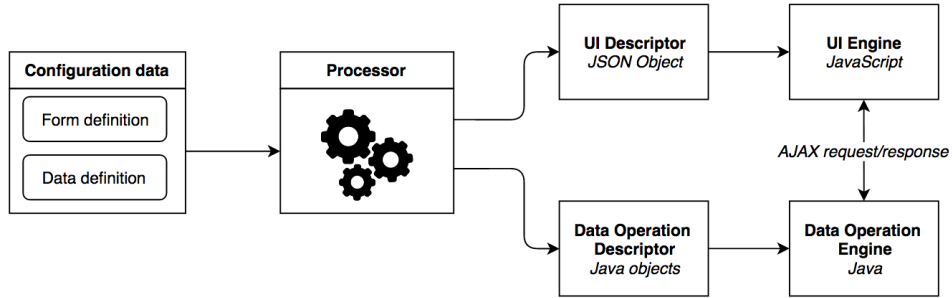


Figure 3.6: Framework functionality outline

Chapter 4

Vocabulary for RDF data input

The aim of this chapter is to show how can different data input problems be expressed through the instantiation of a specific set of Java classes. At first, the elements of the vocabulary is discussed briefly, then the solved two use-cases of the RDFBones project are presented.

4.1 Elements of the vocabulary

4.1.1 Data definitions

The first part of the vocabulary relates to the RDF data definition. The two main classes for this purpose are the *Triple* and the *RDFNode*. Both of them have different subclasses for the custom cases. Figure 4.1 shows the UML class diagram for the different types of node, and Figure 4.2 in turn depicts the classes for the triples.



Figure 4.1: Java classes for RDF nodes

The class diagram does not contain all the fields and methods for the sake of simplicity. The most important field is the *variableName*, based on which the form elements can reference the nodes. In the data definition constant classes can appear, for this stands the class *Constant*. In the vocabulary there are two specific classes for the input nodes. The first is the *MainInputNode*, which represent the variables, that are coming with initial form call. In VIVO these are always the *subjectUri*, the *predicateUri* and the *rangeUri*. The class *FormInputNodes* stands for the variables that appear on the form.



Figure 4.2: Java classes for RDF triples

In the class *Triple*, the three most relevant fields are the *subject*, the *object* and the *predicate*. The first two have the type *RDFNode*, while the predicate is always a constant string. The role of the class *MultiTriple* were already discussed the previous chapter. The *LiteralTriple* plays only a role in the advanced instance selector interface (discussed in more detail later), because currently, the system is not capable of creating literal triples above the labels of the instances. There are two types of restriction triple, one for the classes, the class *RestrictionTriple*, and the other for the instances, the class (*InstanceRestrictionTriple*. The usage of the latter will be discussed in section 4.3. The former is for the restrictions defined with *owl:someValuesFrom* and *owl:allValuesFrom* properties, while its subclass the *QualifiedRestrictionTriple* represents the qualified restriction triples. The last restriction triple, the *GreedyRestrictionTriple* stands for the cases where the query on

the restrictions has to check there are restrictions applied not only directly on the input class, but on its super classes as well.

4.1.2 Form definition

As it was already mentioned in the previous chapter the two main classes for the form layout definition are the *Form* and the *FormElement*. The form has a title as a string and contains the different form elements in its field *formElement*.



Figure 4.3: Java classes for the form

In the class *FormElement* the most important field is the *dataKey*, which holds the variable name for the RDF node it represents. The form element *SubFormAdder* has a field *Form*, which allows the sub form definition. The class *LiteralField* stands for an element, which allows defining the prefix for the labels of all instances created newly through the data input process. Here it is important to note that on the forms implemented for the project, it has to be possible to let the user selecting existing instances, which will contribute to the resulting dataset. The class *Selector* stands for the HTML selector element, which allows the definition selection both of instances and classes on the form. The class *InstanceSelector* denotes such field where the existing instance can be selected through a floating window implemented in JavaScript. This window allows to display more information, by displaying the instances within a table where the different columns contain data about them. This table, its cells, and triples for the additional SPARQL query by

the classes depicted on Figure 4.4. Finally, the class *AuxNodeSelector*, allows the selection of such instances on the form, who does not contribute to the resulting dataset, just they help to filter the instance for the other instance selectors.

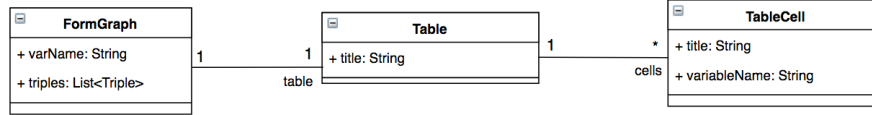


Figure 4.4: Java classes for instance viewer

4.2 Use-cases of the *RDFBones* project

This section covers the two main tasks of the project, the primary skeletal inventories, and the study design execution. These two examples are sufficient to demonstrate the utility of the particular elements in the vocabulary and exemplify how their assembly can lead to a compact definition of complex web application problems.

4.2.1 Primary Skeletal Inventory

Skeletal inventories were already addressed in section 2.9. To be able to understand the software specification of the data input form, it is inevitable to go a bit more into the details of extensions of the *RDFBones* project. The primary skeletal inventory is the main extension, and has the peculiarity that it does not contain any custom bone segments, but refers all of them as entire bones. For this purpose, there is a class called *EntireBoneOrgan*, which has as many subclasses for the particular bones as many bone organ classes have been taken from the FMA ontology. The entire bones are connected to the bone organs through restriction on the property *fma:regional_part_of*. Moreover, for each entire bone organ, a custom completeness datum class is created, which acts as connector between the primary skeletal inventory class and the entire bone classes as well using restrictions.

Figure 4.5 depicts the scheme of the primary skeletal inventory extension. The *X* and *Y* represent the different bone organs classes as examples,



Figure 4.5: Primary skeletal inventory extension

and *E.B.O.* is the abbreviation for entire bone organ, while *C2S* stands for completeness two states. Important to note that the class *CompletenessTwoStatesLabel* has two instances, which are the part of the extension too, and will appear in all skeletal inventory dataset.

The most important skeletal region in the Biological Anthropology Department is the skull, so in the following the data input for the skull will be discussed. The skull does not consist directly of bone organs, but of two sub-subdivisions, so a primary skeletal inventory for a skull is represented with the following triple pattern:

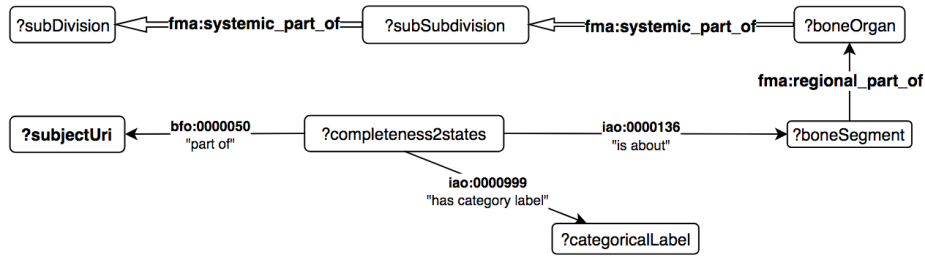


Figure 4.6: Skeletal inventory data triples

The classes of the pattern are not shown because the main scope of the image is to illustrate the necessity of the single and multi triples in the definition. As it was addressed, the subdivision (in our case skull) consists of multiple sub-subdivision, where all consist of multiple bone organs. However, each bone organ has only one bone segment (the entire), and one categorical measurement datum and category label, and belongs to one skeletal inven-

tory. The skeletal inventory is represented by the *subjectUri* variable because the entry form is called from its profile page.

Before showing how the use-case is defined by the descriptor instances, the implemented interface is discussed (Figure 4.7). Under the title of the form, a subform adder element appears. This element is implemented in a way that it incorporates a class selector as well. This is the same if the subform would contain a type selector, just the form layout is more compact. It can be seen the added subform has a title, which is the label of the added class for the element. The subform adder for the bone organs is the same. The subform for the particular bone organs contains the selector through which the user can select if the bone is complete or partly present in the given skull.

Skull

Skeletal Regions **Viscerocranium**

Viscerocranium

BoneOrgans **Left temporal bone**

Left temporal bone **complete**

Neurocranium

BoneOrgans **Right parietal bone**

Right parietal bone **partly present**

Figure 4.7: Generated interface for primary skeletal inventory

Figure 4.8 depicts the complete data definition with the triples and the nodes. For the better readability, the predicates are not denoted, but their value can be found in figure Figure 4.6. The predicate is always *rdf:type* where the subject is an instance and the object is a class. The three rectangles denote the three subgraphs of the graph pattern, divided by the multi triples. The pattern contains the two main input nodes (red - *subjectUri*, *rangeUri*) and three form input nodes (light red - *subSubDivisionType*, *boneOrganType*, *categoryLabel*). The instances without coloring are normal *RDFNode* instances in the descriptor data set, and as they do not appear on the form, they will get a new unused IRI after the submission. The class in-

put nodes get their values based on the SPARQL queries generated from the restriction triples. The classes *completeness2StatesType* and *boneSegmentType* are special because their value will be evaluated after the submission based on the value of the *boneOrganType* node. In this way that the interface hides the complexity of the underlying data model from the user.

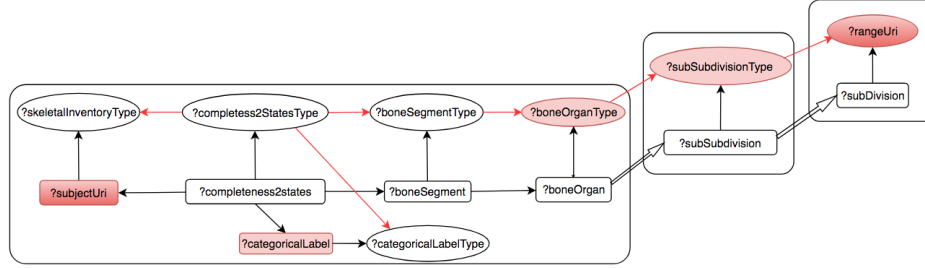


Figure 4.8: Complete data definition

The form layout definition of the Java object is depicted on the Figure 4.9. It can be seen that it reflects the hierarchy of the form, and the form elements refer to the variable names of the nodes in the graph pattern.

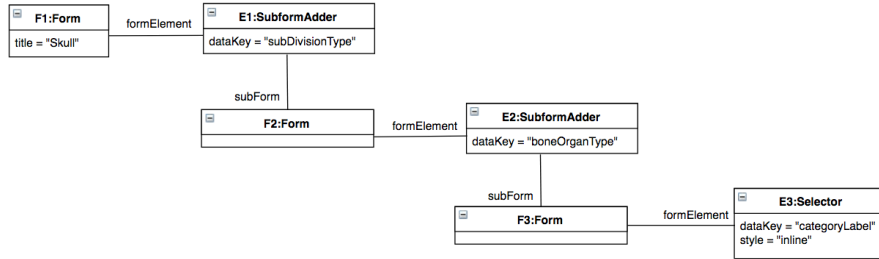


Figure 4.9: UML object diagram for form layout definition

4.2.2 Study Design Execution

The title of the section indicates a class from the OBI ontology. A study design execution is a part of an investigation and consists of several different assays. Assay is a process for assessing a certain quality of a given object. In our case, the input objects are bone segments, and the outputs are information content entities. For example, an investigation aiming to estimate the gender of an individual takes different tokens on a particular skeleton and

defines how masculine or feminine they are, and at the end aggregates the values. Figure 4.10 illustrates an example, the token glabella on the frontal bone, and its expressions grouped into five categories.

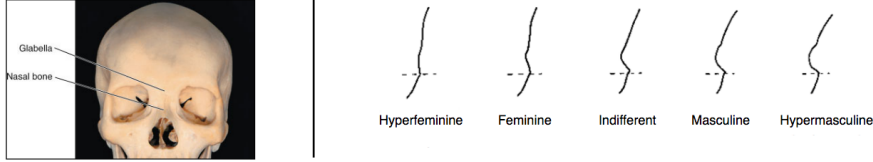


Figure 4.10: Glabella and its expressions

The ontology extensions describing such investigations contains all assays as different OWL classes. The assays are connected with restrictions to specific bone segment and information content entity classes. Since the output data of the assay have to be in chosen from a given set, the extensions contain instances just like by the skeletal inventory the two types of completeness state. Figure 4.11 depict the subset of the *RDFBones* ontology extension for gender estimation, where the expression are represented by the instances of the class *GenderScore*.

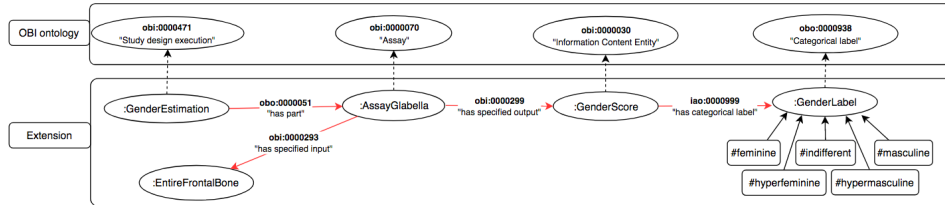


Figure 4.11: *RDFBones* extension for study design execution

The graph pattern of a study design execution can be seen on Figure 4.12. During the execution of one assay, only the assay and the gender score instances will be newly created, because the bone segment is already added to the system through skeletal inventories, and the gender scores are given.

Figure 4.13 shows the implemented interface for the data entry, which is called from the profile page of the investigation instance. The two first elements are additional with respect to the interface discussed in the previous section. The first is an auxiliary node selector, whose purpose is to help the user in finding the input bone segments by selecting the skeletal inventory

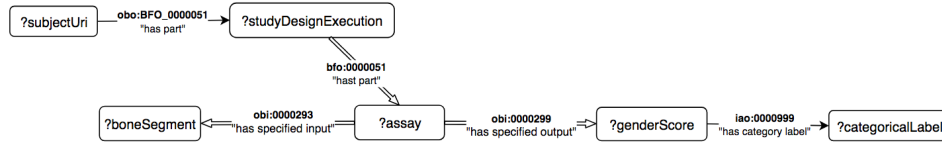


Figure 4.12: Input form for study design execution

they belong to. The second is a literal field for setting the prefix for the label of the newly created instances, which facilitates later the identification of the performed assays.

Figure 4.13: Input form for study design execution

The interface allows to add multiple measurements, but in our case, only the gender score (sex score) is defined in the extension, but there may be such advanced assays which have multiple output data. The bone segment is an instance selector, where the gray button with the *Select* label opens the floating window for more elaborate instance selection (Figure 4.14). Such window allows the selection of multiple instances, which is necessary because there are such tokens, which refers to multiple bone segments at the same time.

The complete RDF graph pattern can be seen on Figure 4.15, where the green nodes are the constants, and the dotted red arrows are instance restriction triples. The framework knows that the instances which are connected with instance restriction triples are not part of the created data. The interface description works the same way as by the skeletal inventories through

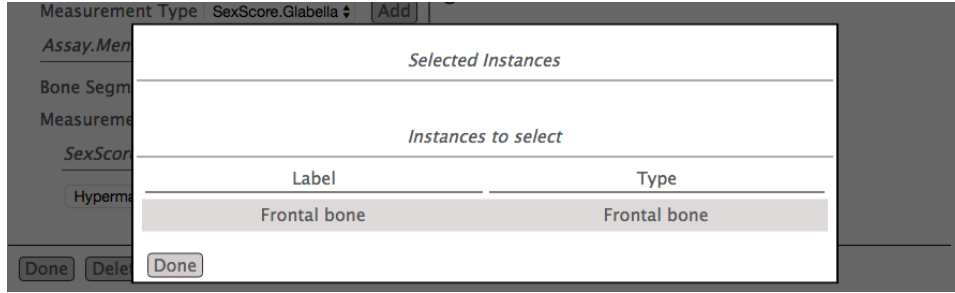


Figure 4.14: Instance selector for existing bone segment

Java objects.

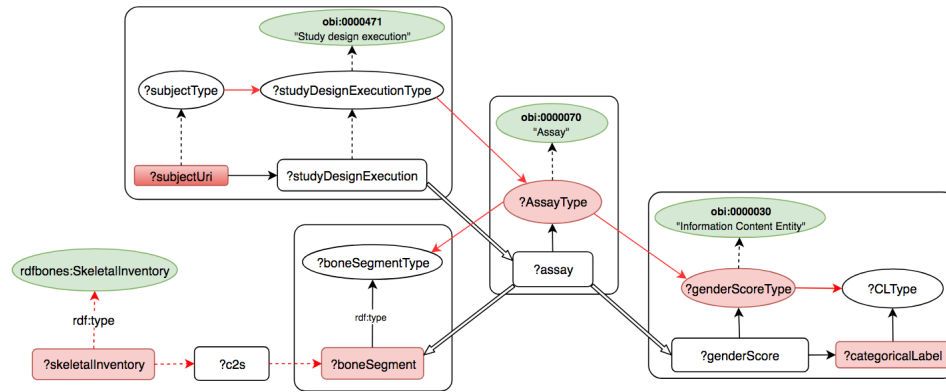


Figure 4.15: Complete data model

The two discussed use-cases showed that the form scheme is defined by the designed vocabulary, which representation of the extension scheme. So the designed system offers a language to build web application rapidly on the top of different ontology extension schemes.

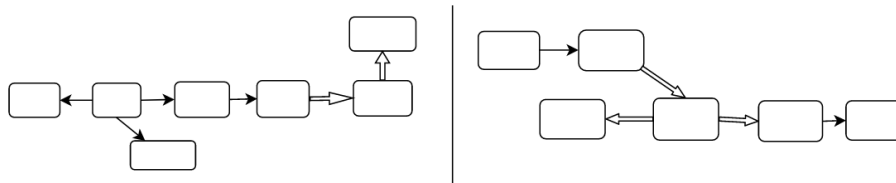


Figure 4.16: Extension schemes of the discussed use cases

Chapter 5

Framework functionality

The aim of this chapter is to present the main mechanisms of the implemented software framework, which is capable of operating on a high-level configuration data. Section 5.1 contains a more abstract description of the functionality, while section 5.2 goes into the implementation details both on the server and client side programming, including how the framework can be integrated into the applied VIVO framework. In both sections, the explanations mostly refer to the use-cases discussed in the previous chapter.

5.1 Main software modules and tasks

On Figure 5.1 the workflow and the main components of the framework can be seen. After the initial form request, an algorithm processes the configuration data and generates form, dependency and graph objects for the client and the server.

This section is split into three subsections. The first (section 5.1.1) discusses the processing of the configuration data and the generation of the functionality descriptor objects. The second part (section 5.1.2) is about the client functionality including the asynchronous communication with the server. Finally, the third (section 5.1.3) part covers the process after the submission, namely how the RDF data is generated based on the data coming from the client, as well as how the existing form data is retrieved from the triple store for editing.

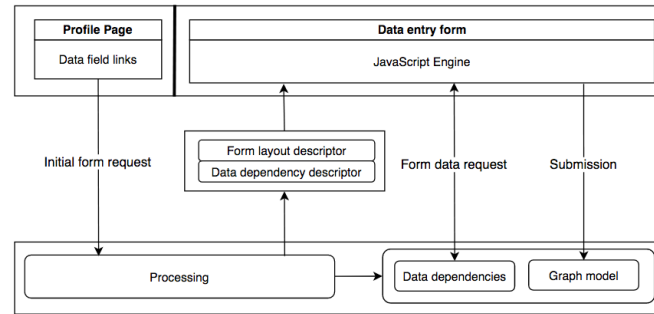


Figure 5.1: More detailed scheme

5.1.1 Validation

The processor algorithm has four main tasks to solve. The validation of the configuration data, the generation of the form descriptor JSON object, and the Java objects for the data dependencies and the graph model.

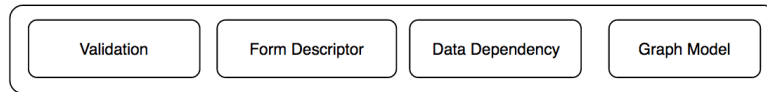


Figure 5.2: Processor tasks

The input of the algorithm is a set of triples describing the data model with its constraints, and the form model, which refers to the nodes in the triple set. The first task to do is the validation because the descriptors are not supposed to be generated based on incorrect configuration data. The validator process has two scopes of the checking, the nodes themselves and the graph structure built by the triples.

Figure 5.3 depicts an example data model and illustrates some cases of valid and invalid nodes. The red nodes are coming as input, and the green stands for a constant. The explanation starts with the discussion of class nodes (ellipses). Node 2 is valid because it is possible to generate a SPARQL query that retrieves the possible values of it for the form. The query contains one triple which asks the subclasses of the constant class. Furthermore, node 1 is valid as well, because there is path to it from a valid class node, therefore there is again a SPARQL query for its values after the submission. However,

node 3 is not valid because it does not contribute to any path with valid input node or constant. Here it is important to note that the path cannot come from the instance to the class, just the other way around. So the path $2 \rightarrow 1 \rightarrow 4$ counts in the processor routine, but path $2 \rightarrow 5 \rightarrow 6 \rightarrow 3$ does not.

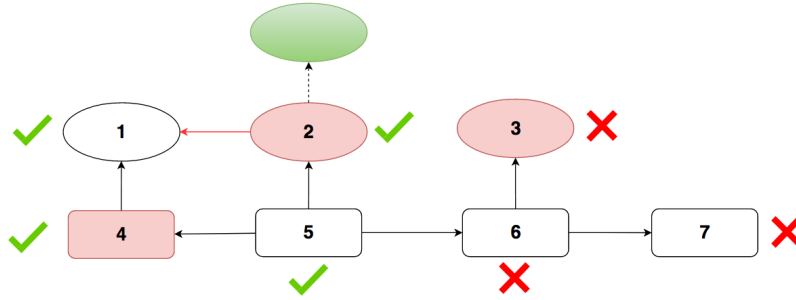


Figure 5.3: Valid and invalid nodes

The next task above the classes is to check if each instance (rectangles) has a value or type by the RDF triple creation. The instances coming from the interface are automatically valid because their URI is an input value. But the ones that have to be generated newly and get a new unused URI as value must contribute to a triple as subject, where the predicate is *rdf:type* and the object is a valid class. Consequently, node 5 is valid since its type class is valid, but node 6 is not. Finally, node 7 is not valid, since it does not have any type class defined for it.

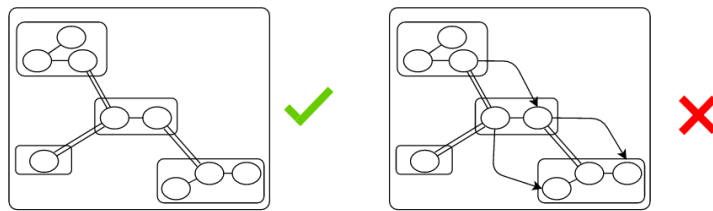


Figure 5.4: Valid and invalid graph

Above the nodes, the whole graph itself have to be investigated too. In the previous chapter, the triple type *MultiTriple* (double line on the figure) were introduced. The rule regarding this type of triple that it divides the whole graph into subgraphs and the subgraphs are allowed to be connected to

each other only by these triples. The reason is that only to this hierarchical graph scheme can the form JSON object be mapped. Figure 5.4 illustrates the valid and invalid graph arrangements.

5.1.2 Dependencies and form functionality

The set of triples describing the data model builds a graph where the different input nodes are connected to each other. A further task of the processor algorithm is to determine the subgraphs that define the SPARQL queries for the nodes appearing on the form. The elements on the form have a specified order and a variable can be dependent only of the main input nodes, or of such nodes that are before them on the form.

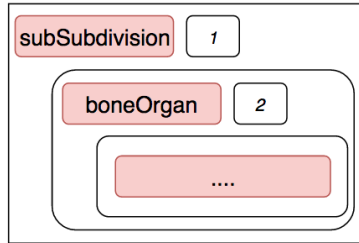


Figure 5.5: Form element order

Figure 5.5 shows a simplified form layout for skeletal inventories and emphasizes the order of the elements with a number. The node *subSubdivision* can only be dependent of the main input nodes because it is the first form one. However, the node *boneOrgan* in turn can be dependent on the *subSubDivision* as well, because its value has to be set before the selector options in the subform are loaded.

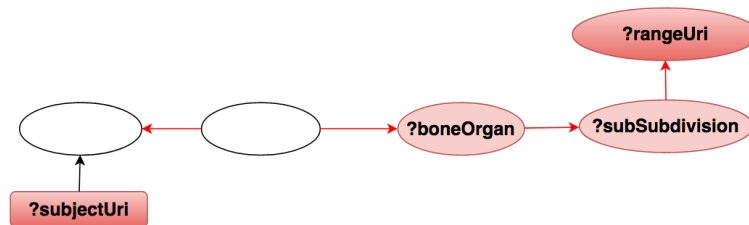


Figure 5.6: Skeletal inventory data constraints

To exemplify the dependency generator algorithm Figure 5.6 depicts the

restriction triples of the skeletal inventory use-case. The task is to get one or more paths to particular form variables, where the start nodes or the path are main input nodes or former form node. As the form's first element refers to the node *subSubDivision* its dependency has to be evaluated first. Since the node *subSubDivision* contributes to two restriction triples in the data scheme it is necessary to check the both paths. Since this node cannot be dependent on any other form node, it is dependent on the two main input nodes (*subjectUri* and *rangeUri*). For the node *boneOrgan*, the *subjectUri* will be the input of the left path again, but on the right path the algorithm terminates by the *subSubdivision* variable. Figure 5.7 shows the results subgraphs of the algorithm, where the output is depicted with green and the inputs with red and light-red respectively.

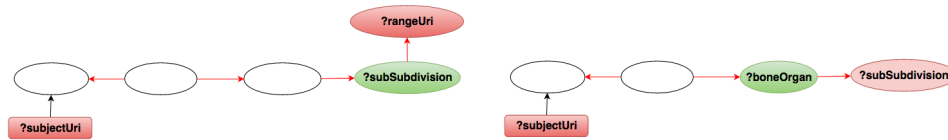


Figure 5.7: Form dependency subgraphs

The result of the processing is important both for the descriptor of the client and the server. On the server, the dependencies are saved using such classes that have the necessary fields for the paths, and for the output and input nodes. The initialized dependency descriptor objects are collected in a data map, where the key is the variable name of the output node. For the client, the dependency descriptor is a JSON object that contains which form node is an input for the individual variables in an array. The task of the form by loading new subform is to check this array and get the variables required values from the form.

```
var dependencies = {
  subSubDivision : [],
  boneOrgan : [ "subSubDivision" ]
}
```

Listing 5.1: Dependency descriptor JSON for skeletal inventories

Above the dependency the descriptor the JavaScript framework obtains

the layout descriptor JSON for the form elements upon which it can generate the form. The mechanism of the form descriptor generation is quite simple because it is practically the conversion of a Java object into a JSON object. The fields of the Java object appear in the JSON objects as key. All individual Java class representing a form element has its type field value keyword (i.e "literalField" or "subFormAdder"). Based on this value can the form generator algorithm call the appropriate JavaScript class.

```
public JSONObject getDescriptor(){
    JSONObject object = new JSONObject();
    object.put("title", this.title)
    object.put("type", this.type)
    ...
    return object;
}
```

Listing 5.2: Java to JSON

Moreover, if the form element is sub form adder then it has a field *subForm*, which comes well of course into the descriptor, and this is the way how the multi-level JSON object is created.

```
object.put("subForm", this.subForm.getDescriptor());
```

Listing 5.3: Subform descriptor

Figure 5.8 illustrates a generated JSON object for a form with the data dependencies too.

5.1.3 Graph model generation

The previous section outlined how the set of Java objects are converted into the form descriptor JSON object, and into Java objects describing the variable dependencies for the AJAX requests. A further task of the framework is to receive the submitted multi-level JSON object coming from the client and generate the appropriate set of RDF triples. So in order to prepare the server for the reception of the form data, the same object structure have to be generated, which is coming from the form. We have seen in the previous

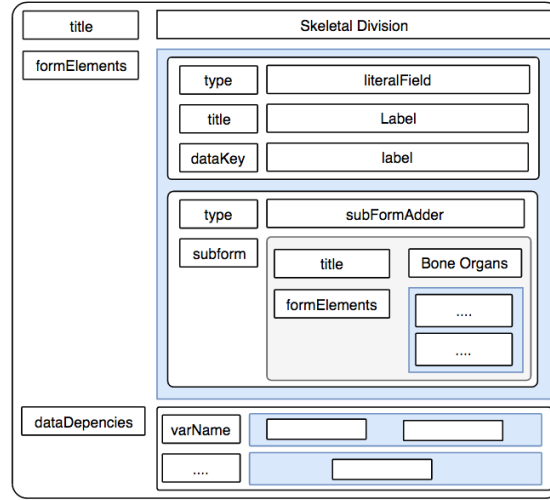


Figure 5.8: Form descriptor JSON

chapters, that the form data object has the same scheme as the form has, and the form follows the scheme defined by the multi triples in the triple set. Therefore the task of the last part of the processor algorithm is to decompose the set of triples by multi triples into graphs. The graph structure is represented in the server by a Java class called *Graph*.

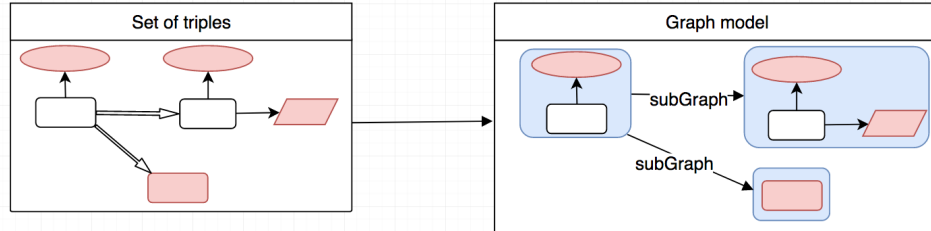


Figure 5.9: Conversion from triples into graph model

The decomposition starts by the initial RDF node, which defines the main graph. The class *Graph* has a `Map<String, Graph>` field where the subgraphs are stored. The keys of the map are equivalent to the keys of the arrays in the incoming JSON object. The other keys of the JSON are the variables of the corresponding graph. The following code snippet shows the basics of the RDF data generation from multi-level JSON object.

```
saveData(JSONObject formData){  
    this.save(formData);  
    for(String keys : this.subGraphs.keys()){  
        JSONArray array = formData.get(key);  
        Graph subGraph = this.subGraphs.get(key);  
        subGraph.saveArray(array)  
    }  
}
```

Listing 5.4: Saving routine in Java

The save routine creates the RDF triples of the graph based on the data fields of the JSON object. The loop iterates through the subgraphs of the graph and gets the arrays with the same key from the JSON and passes it to the corresponding subgraphs, that perform the same algorithm as many times, as many elements of the input array have. This is the way how the multi-dimensional JSON is processed by the same structure of graph model on the server.

FiguregraphProcess illustrates the process of the JSON-RDF conversion by means of graph model. The advantage of this graph model that it can be applied the same way for the data retrieval, where the graph performs the SPARQL queries based on its triples, and generates the arrays of objects from the result table.

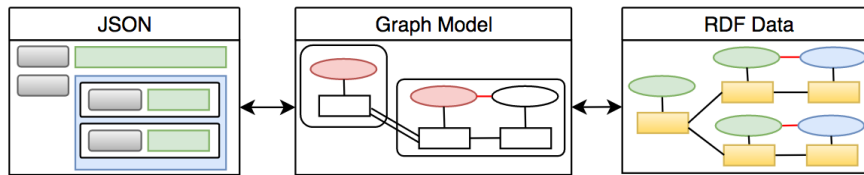


Figure 5.10: JSON/RDF through the graph model

5.2 Implementation

The description of the implementation starts with the client side because it is more independent from the other parts, and it sufficient to understand the functionality of the server accurately.

5.2.1 Client side

This subsection presents the basics of the JavaScript implementation that realizes the dynamic form generation and event handling based on JSON form configuration data. The first part covers the creation of a form itself, and how the data is set to form data object, while the second part is about how the form enables multi-dimensional data input by means of subform adders.

Form loading

In contrast to Java, JavaScript codes are not necessarily built up in an object-oriented manner. On pages where the elements are statically defined in HTML, it is sufficient to assign event handler routines to them. However in our case, none of the elements of the page is coded into HTML, but everything is dynamic and thus added by JavaScript. In the implementation, the JavaScript classes are applied, whose input is the descriptor object, based on which they generate the corresponding data input fields, and handles the data entered through them by the user. In this section the functionality of the two main classes, the *Form* and *Formelement* is discussed.

Figure 5.11 illustrate the structure of the two main classes. The most fundamental difference between these JavaScript classes to Java classes, that they do not contain only fields and routines (or methods) but eventually UI elements as well. The UI elements can represent an HTML tag and can be added or removed any time by the routines. Each class of the implemented JavaScript library has a defined set of UI elements.

The form generation process begins with the initialization of a *Form* object, where the constructor (like in Java) gets the descriptor JSON object coming from the server. As it was described in the previous chapter, the descriptor contains a list of the form element descriptor objects. The *load-FormElements()* routine iterates through on this array, and initiates the form

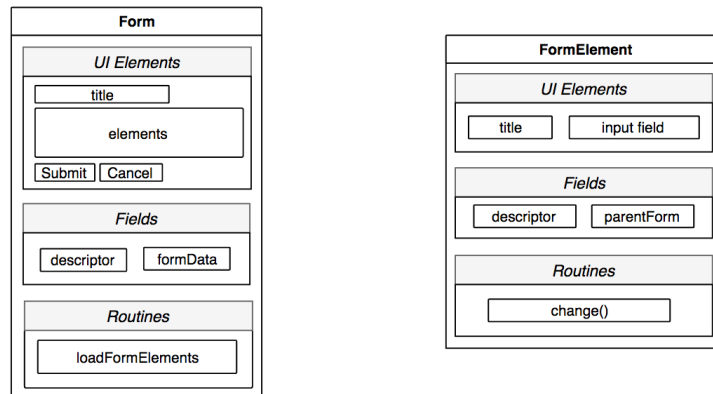


Figure 5.11: Form and form element JavaScript classes

element objects.

```

var formData = new Object()
for(var i = 0, i < formElements.length; i++){
  switch(formElements[i].type){
    case "literalField":
      var element = new LiteralField(formElements[i], formData)
      break;
    case "selector" : ...
  }
  this.elements.append(element.container)
}

```

Listing 5.5: Form generation based on configuration data

Each form element type is represented as subclass of the *formElement* class. They all have a container UI field, which contains their title and input field HTML element. This container field is added to the *elements* field of the *Form* object.

Listing 5.6 show a small cut from the code of the *LiteralField*, which is the subclass of the *FormElement* class. The field *inputField* is the HTML `<input/>` tag, and if its value changes then the *editHandler* routine is called. The *editHandler* is the function that realizes the dynamic form data creation, by setting the value of the input field into the form data object with the key defined in its the descriptor. The key is stored in the *dataKey* field of the descriptor, which is the variable name of the *RDFNode* the input element

represents.

```
class LiteralField extends FormElement{
  constructor(descriptor , formData){
    super(descriptor , formData)
    this.inputField = $("<input/>").type("text").change(this.
      editHandler)
    ...
  }
  editHandler(){
    this.formData[this.descriptor.dataKey]=this.inputField.val()
  }
}
```

Listing 5.6: Form element

This is the fundamental mechanism of how object-oriented JavaScript can be employed to generate forms and put the entered values into JSON object based on configuration data.

Sub forms

The previous section explained how the form algorithm creates the JSON object of the form data. This section extends the explanation of how it is possible to add the multi-level data by sub form adders. To this two new JavaScript class functionality is outline, the *SubFormAdder* and the *SubForm*. The former is the subclass of the *FormElement* and the latter of the *Form* class.

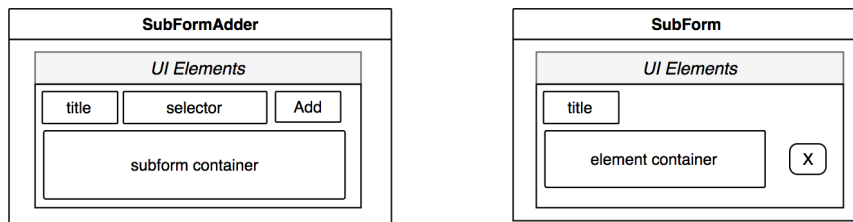


Figure 5.12: SubForm and SubFormAdder JS classes

Figure 5.12 depicts the UI elements of the two classes. The routines and fields are inherited from the parent classes. The class *SubFormAdder* has a

button, which lets the user add new subforms, which are appended into the subform container. The class *SubForm* has additionally to the parent class a delete button for the cases if the user wants to delete the added dataset.

Listing 5.12 shows the relevant part of the code in the class *SubFormAdder*. The essence of the class is that the constructor initiates an array (with `[]`) in the form data object, to which the subform data object will be added dynamically upon the click events. So if the user clicks the add button, then a further object is initialized (*subFormDataObject*), which will be the data object of the subform. Important to note that this object will contain the value of the selected option of the subform adder with the key defined in the *dataKey* field of the descriptor. After the initialization of the object, it is pushed to the array, and the new *SubFormAdder* instance is created, whose container is appended to the subform container of the subform adder.

```
class SubformAdder {
  constructor(descriptor, formData){
    this.addButton = $("").text("Add").click(this.add)
    this.subFormDescriptor = this.descriptor.subForm
    this.formData[this.descriptor.predicate] = []
  }
  add(){
    var subformDataObject = new Object()
    subformDataObject[this.descriptor.dataKey] = this.selector.
      val()
    this.formData[this.descriptor.predicate].push(
      subformDataObject)
    this.subFormContainer.append(
      new SubForm(this, this.subFormDescriptor,
        subformDataObject).container)
  }
}
```

Listing 5.7: Sub form adder routine

The class *SubForm* works almost the same way as its parent, but with the difference that it checks if there is such selector among its elements, whose data has to be loaded dynamically through AJAX because its value is dependent on one or more previously set elements of the form.

5.2.2 Server side

This section contains the discussion of the main elements of the implementation on the server. On the client side, the JavaScript files containing the classes and libraries were just included into the generic template file, but the server side integration into VIVO is a more complicated issue, therefore the first subsection is dedicated entirely to it. Furthermore, the handling of the AJAX calls for form data with the variable dependencies are presented here, as well as the routines how the graph model can save, edit and retrieve the data.

VIVO integration

In VIVO, the form loading starts from the property fields of the profile pages, Based on the property IRI the generator class is retrieved from the configuration triples (Figure 2.13), and the generator class returns an instance of *EditConfigurationVTwo* class. The approach implemented in the frame of the thesis does not replace this scheme but makes it simpler. The idea is that the complete definition is stored in one object called *FormConfiguration* (listing 5.8).

```
class EditConfigurationVTwo {  
    ...  
    FormConfiguration formConfig;  
    ...  
}
```

Listing 5.8: Added field to the VIVO configuration class

Figure 5.13 shows the UML class diagram *FormConfiguration* class. The first field (*connector*) connects the server implementation classes to the triple store (Figure 5.14). This has to be implemented individually for VIVO. The *mainGraph* field is the data definition, the *form* is for the form, and the *dependencies* and *graphMap* are used by the AJAX calls.

Figure 5.15 shows the process of loading of a custom entry form in VIVO. The first step is to find the generator class based on the value of the *predicateUri* parameter of the initial request. If the class has been found, the processor algorithm is performed, and the necessary JSON and Java object

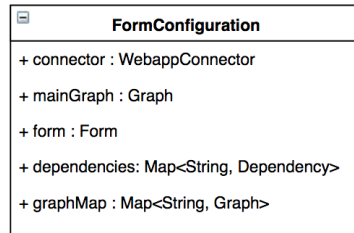


Figure 5.13: UML class diagram for FormConfiguration

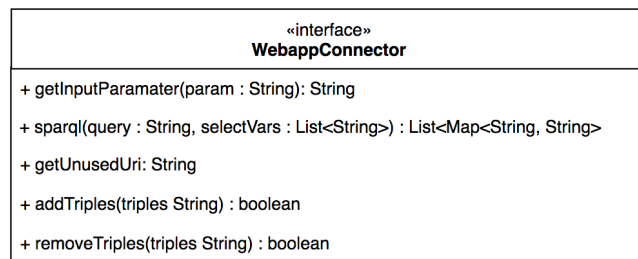


Figure 5.14: UML class diagram for the WebappConnector interface

are generated. Afterwards, the server saves the form configuration object in its cache with a key, that is forwarded to the client in the *editKey* JavaScript variable. The box in the middle of the image shows that the response web page includes the JS libraries (simplified notation *framework.js*), and the value of the *editKey*. This is the value will be sent with each AJAX request to the server, and based on this parameter the can the server find the form configuration instance that returns the required JSON objects.

Upon the AJAX calls from the form, the class *AJAXController* is called, which gets the found form configuration instance the incoming JSON object (*requestData*). Each AJAX call has a parameter *task*, based on which the framework decides which operation has to be completed.

```

setResponse(FormConfiguration formConfig, JSONObject request) {
    switch (request.get("task")) {
        ....
    }
}
  
```

Listing 5.9: AJAX request server routine

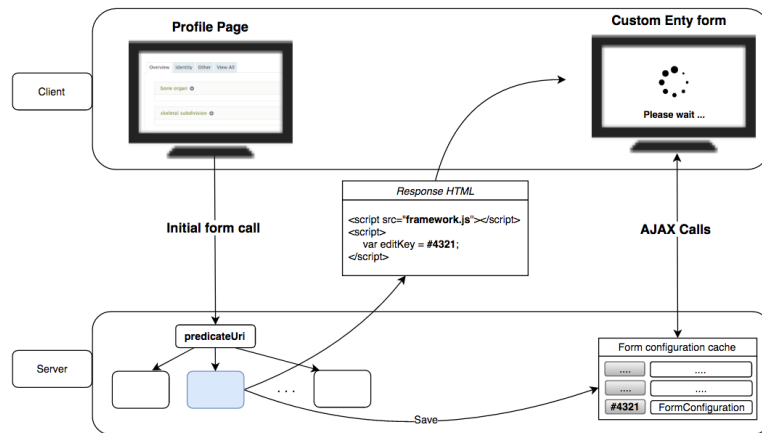


Figure 5.15: Form loading process

Form data loading

Important part of the framework functionality is the loading of options of particular selectors on the forms. As it was mentioned, the variables on the form can be dependent on other values of the forms. The request has three main parameters :

```
var formDataRequest = {
  editKey : "93181",
  task : "formData",
  variableToGet : "boneOrgan",
  inputParameters : { subSubDivision : "fma:5058" }
}
```

Listing 5.10: Example request JSON for form data

The AJAXController then finds the variable dependency instance from the *dependencies* map based on the *variableToGet* field:

```
switch (request.get("task")) {
  case "formData":
    return dependencies.get(request.get("varToGet")).getData(
      request)
}
```

Listing 5.11: Loading form data from FormConfiguration

Figure 5.16 shows the UML diagram of the variable dependency. The most important field is the *dependencies*, which is a set of triples describing the path from the input variables to the output variable. The method *getData* returns the JSONObject containing the list about the options.

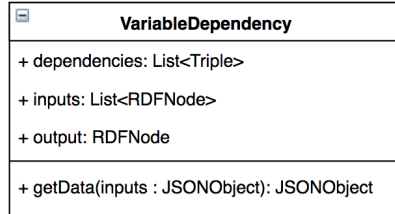


Figure 5.16: UML class diagram for VariableDependency

In case, if the dependency is only one restriction triple, and there is only one input variable, the variable dependency class generates i.e. the following query.

```

SELECT ?outputVar ?label
WHERE {
  ?inputVar      rdfs:subClassOf      ?restriction .
  ?restriction   owl:onProperty      fma:systemic_part_of .
  ?restriction   owl:someValuesFrom  ?outputVar .
  ?outputVar     rdfs:label           ?label .
  FILTER (?inputVar = fma:5058)
}
  
```

Listing 5.12: SPARQL query generated by class restriction triple

Saving, editing and retrieval of RDF Data

The graph pattern of the certain input forms is stored in the graph object. The graph class has the necessary fields, which were initialized during the processing. Based on its fields the graph objects can cope with the incoming data and can retrieve the existing ones.

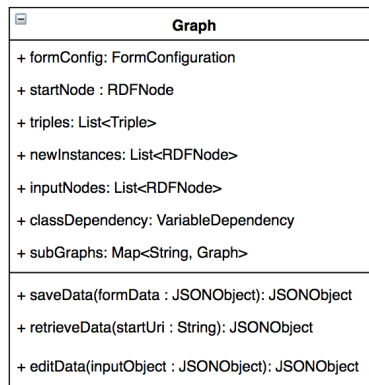


Figure 5.17: UML class diagram for Graph

For the saving and retrieving the whole dataset is addressed, therefore the main graph is called, but if a particular subform is edited on the form the graph, whose data is changes retrieved from the *graphMap* map of the form configuration.

```

switch (request.get("task")) {
...
case "saveData":
    return mainGraph.saveData(request)
case "retrievedData":
    return mainGraph.getData(request)
case "editData":
    return graphMap.get(request).editData(requestData)
...
}
  
```

Listing 5.13: Data operations in the AJAXController

Chapter 6

Conclusion

6.1 Evaluation

The applied VIVO framework offered the possibility to define data input processes in a declarative way to some limited extent by defining the elements of the input form and the RDF graph pattern. The simplicity lied in that VIVO allowed the setting of literals of particular instances, which required only static HTML forms and simple value substitution algorithms. However in the *RDFBones* project, the emphasis was not on the literals but the RDF instances, and there were such cases where multiple instances had to be created through one data entry form. This required dynamic web pages with handler routines instead of static form elements and the server routines had to become more complex too. Important challenge moreover that rules regarding which entity belongs to which are declared in ontology extension, and these definitions had to be considered by the interface generation.

During thesis, the VIVO idea was further developed, so that the system can cope with the more complex problems. To achieve that the individual cases can be solved without coding rapidly, an extended vocabulary were designed that can express the problems of multi-level data RDF input, a code library was developed for the client and the server that were able to manage the advanced functionality based on a certain problem descriptor data set. In the vocabulary related to the data definition the most important advancement is that it is possible to express if a subject and the object of a particular triple in the graph pattern are in one-to-one or in one-to-many relationship

with each other (*Triple* vs. *MultiTriple*). On the form definition, this cardinality related definition is reflected by the subforms (*SubformAdder*). These elements allowed the expression of forms and data processor routines that can handle multi-dimensional dataset. A further improvement that the vocabulary for data definition does not relate only to the RDF triples that were supposed to be stored, but to RDF triples as well which contained the description of the system, namely the OWL restriction in the ontology extensions (*RestrictionTriple*). The utility of this definition that it connects RDF nodes that were represented on the interface, thus the dependency between form elements could be expressed as well in a declarative way. From the restriction triples, the appropriate SPARQL queries are generated, and the client and server algorithm together through AJAX calls could realize the adaptive interface. Finally, we have seen that the resulting RDF dataset does not necessarily consist only of new instances, but existing ones as well. The developed vocabulary is able to express these cases, and make possible the convenient browsing and selection of them with further form elements and constraints (*AuxNodeSelector*, *InstanceSelector*, *InstanceRestrictionTriple*).

The main benefit of the system that it abstracts from the low-level implementation details and the developer does not have to care about how the data created, edited and deleted by the application. It is sufficient to think about the scheme, the constraints and the mapping to the interface of the data describing a particular entity, and the generic client and server libraries realize the data flow between the user and database. The system can be applied then for any problems where the rules of the system lie in ontological statements. Moreover, due to the dynamic VIVO profile page, which allows the discovery of the RDF data graph, the created instances can be browsed without additional programming, and their literal values can be set through custom entry forms developed with the original VIVO framework tools. So the implemented system embedded into VIVO offers a widely employable Semantic Web-based data management application.

6.2 Future work

Improved handling of OWL restrictions

Currently, the implemented framework cannot handle scalar measurement data, only categorical data, which is represented by RDF instances in the extensions. But there are such subprocesses of the investigations where the output data is some scalar value or even a string. In this case, the restriction points to some class of the XML schema representing a literal type. This means regarding the user interface that the type of the particular data input fields is not given directly in the form descriptor, but it was retrieved from the ontology extension. The handling of such cases would lead to more even more adaptive data input forms.

The data input process descriptor vocabulary allows the handling of different types of restrictions, like *owl:allValuesFrom*, *owl:someValuesFrom* and qualified cardinality restrictions. However, currently all of them are considered just as connectors between classes, not as exact data rule descriptors. Thus the expressiveness of the OWL vocabulary was exploited only to a very limited extent. In the future, the server should send not only the list of the classes as options of the particular selectors but as well how many of the individual pieces have to be added at least or most from them. Then form handler routines could ensure that only such dataset is created that fulfills the cardinality constraints as well.

RDF data based form configuration

Currently, the data input problem descriptor vocabulary is implemented in Java classes, and thus the descriptor data is in the form of Java objects. Therefore application programming still needs the creation Java files and writing some tiny Java code for the object instantiation. In the future, the vocabulary will be implemented as an OWL ontology. This solution would mean the advantage that it would be possible to build RDF data input forms entirely the same way just for the descriptor RDF data input. Thus the implemented framework could be programmable completely through the interface since VIVO profile pages for data display can be configured as well through RDF triples.

Erklaerung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum

Unterschrift

Bibliography

- [1] Felix Engel, Stefan Schlager, and Ursula Witwer-Backofen. An infrastructure for digital standardisation in physical anthropology. 11, 04 2016.
- [2] Katy Börner, Michael Conlon, Jon Corson-Rikert, and Ying Ding. *VIVO: A Semantic Approach to Scholarly Networking and Discovery*. Synthesis Lectures on the Semantic Web. Morgan & Claypool Publishers, 2012.
- [3] Markus Lanthaler, David Wood, and Richard Cyganiak. RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C, February 2014. <http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/>.
- [4] Dan Brickley and Ramanathan Guha. RDF vocabulary description language 1.0: RDF schema. W3C recommendation, W3C, February 2004. <http://www.w3.org/TR/2004/REC-rdf-schema-20040210/>.
- [5] Mike Dean and Guus Schreiber. OWL web ontology language reference. W3C recommendation, W3C, February 2004. <http://www.w3.org/TR/2004/REC-owl-ref-20040210/>.
- [6] Cornelius Rosse and José L.V. Mejino Jr. A reference ontology for biomedical informatics: the foundational model of anatomy. *Journal of Biomedical Informatics*, 36(6):478 – 500, 2003. Unified Medical Language System.
- [7] Anita Bandrowski, Ryan Brinkman, Mathias Brochhausen, Matthew H. Brush, Bill Bug, Marcus C. Chibucos, Kevin Clancy, Mélanie Cour-

tot, Dirk Derom, Michel Dumontier, Liju Fan, Jennifer Fostel, Gilberto Fragoso, Frank Gibson, Alejandra Gonzalez-Beltran, Melissa A. Haendel, Yongqun He, Mervi Heiskanen, Tina Hernandez-Boussard, Mark Jensen, Yu Lin, Allyson L. Lister, Phillip Lord, James Malone, Elisabetta Manduchi, Monnie McGee, Norman Morrison, James A. Overton, Helen Parkinson, Bjoern Peters, Philippe Rocca-Serra, Alan Ruttenberg, Susanna-Assunta Sansone, Richard H. Scheuermann, Daniel Schober, Barry Smith, Larisa N. Soldatova, Christian J. Stoeckert, Jr., Chris F. Taylor, Carlo Torniai, Jessica A. Turner, Randi Vita, Patricia L. Whetzel, and Jie Zheng. The ontology for biomedical investigations. *PLOS ONE*, 11, 11 2015.

- [8] Georg Hohmann and Mark Fichtner. Embedding an ontology in form fields on the web. In *Semantic Web Journal - Interoperability, Usability, Applicability*, vol. 3, no. 4, pp. 1-9, 2012, 2012.
- [9] Rafael S. Gonçalves, Samson W. Tu, Csongor I. Nyulas, Michael J. Tierney, and Mark A. Musen. Structured data acquisition with ontology-based web forms. In *Proceedings of the International Conference on Biomedical Ontology, ICBO 2015, Lisbon, Portugal, July 27-30, 2015.*, 2015.
- [10] *Definition of the CIDOC Conceptual Reference Model*, 5.0.1 edition, 2009. March 2009.
- [11] M. Hanus. Type-oriented construction of web user interfaces. In *Proc. of the 8th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'06)*, pages 27–38. ACM Press, 2006.
- [12] Michael Hanus. *A Functional Logic Programming Approach to Graphical User Interfaces*, pages 47–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.
- [13] Bernd Braßel, Michael Hanus, and Marion Müller. *High-Level Database Programming in Curry*, pages 316–332. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.