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
Examiner
Prof. Dr. Georg Lausen
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 on a high-level declarative way that abstracts from the low-level implementation details. 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

Kurzfassung auf Deutsch

Contents

List of Figures

List of Tables

Chapter 1

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 by means of 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 programmed in generic way so that it adapts its interface to the various cases. During the project an open-source web application framework, called VIVO, is adopted and developed.

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 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 kind of domain, where the rules are defined by means of 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 dedicated to the discussion of the functionality of the developed software framework. Its first section provides an overview about 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.

The three most important types of node are the instances, classes and literals. An instance represents concrete a entity like person, 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 by means of 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 a subjects of a triple.

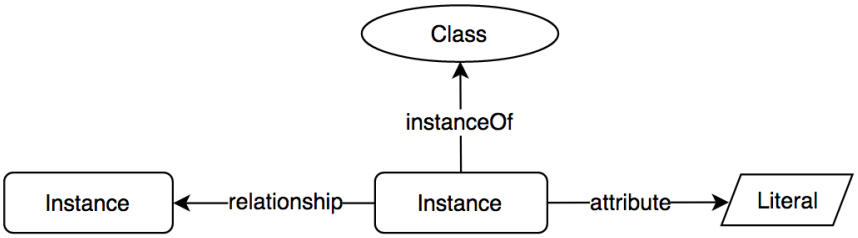

Figure 2.1: RDF graph notation

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

The provided example gave just an insight into 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 a 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 own namespace, which are for the sake of readability often abbreviated with prefixes. In such cases the IRIs

are represented with 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 do not have to 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 with 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 offers classes and properties that allows to express the rules of the RDF data. 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.

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 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 vocubulary 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

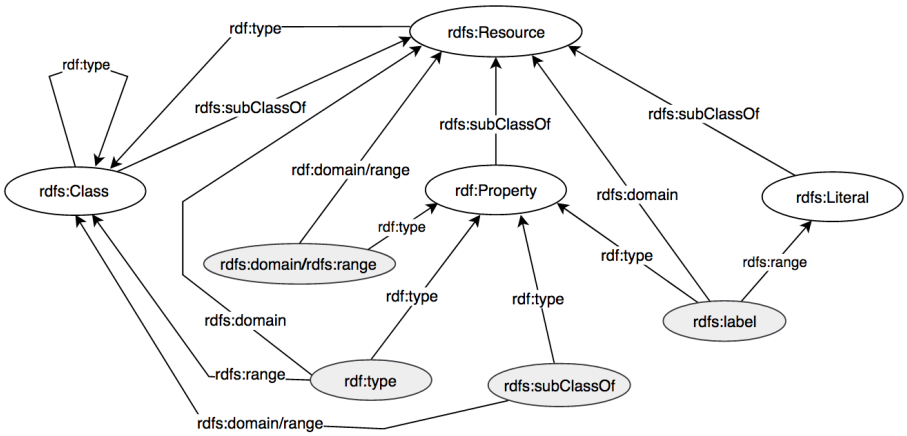

Figure 2.2: Class hierarchy of the RDF/RDFS vocabulary

because it is the instance of itself, and a resource at the same time because it is subclass of *rdfs:Resource* class. This why every instance and class is a resource in RDF. The grey ellipses are the properties, and for each their domain and range are defined. Finally, the *rdfs:Literal* class is the superclass (inverse term for subclass) of all XMLS datatypes.

2.1.3 OWL

Ontology Web Language (OWL) is an extension of the RDFS vocabulary [3]. OWL offers several additional useful classes and properties, that allows more fine grained data modeling. OWL has its own namespace, which is conventionally denoted with the *owl* prefix:

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
```

Listing 2.6: OWL namespace

Figure 2.3 shows a subset of the OWL vocabulary, that is relevant for the thesis. OWL extends the *rdf:Property* with two subclasses. The *owl:ObjectProperty* represents the properties whose range is a class, while the *owl:DataTypeProperty* is for those properties where the range is literal data type. OWL's most important capability is to express restrictions on the data. Instead of depicting the range and domain classes of the particular properties in the OWL vocabulary, the Figure 2.4 shows the scheme of the two types of restriction used

in our project.

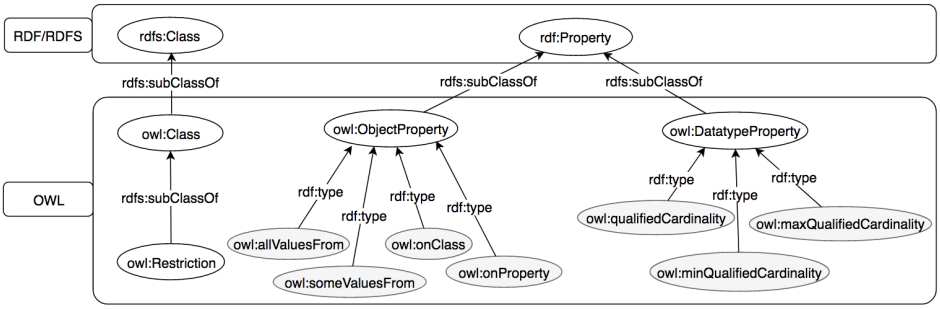

Figure 2.3: A subset of OWL vocabulary

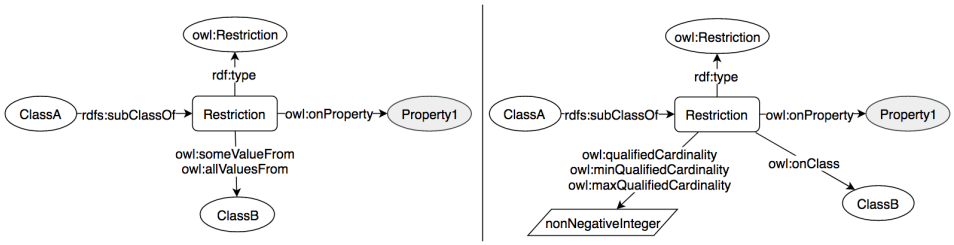

Figure 2.4: Scheme of the restrictions

On the left side the scheme of the value restriction, and on the left side the scheme of the qualified cardinality restriction is depicted. The triples with multiple predicates on the figure show the possible properties that can be applied for that triple. Important by both cases that the central restriction instance (rectangle) is not an RDF resource but a blank node. Blank nodes are such RDF nodes that does not have a IRI. Later in this section example will be provided regarding their definition.

The idea is that the restriction node encompasses the triples that describing a certain rule of the domain. The class on which the restriction is applied, is the subclass of the restriction blank node, while the property is connected to it with the *owl:onProperty* property. By the value restriction the *owl:someValuesFrom*/*owl:allValuesFrom* properties define the instances of *ClassA* can constitute a triple with property *Property1* at least with one instance/and only with instances of *ClassB*. The qualified cardinality restriction asserts how many instances have to present at least/exactly/at most

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

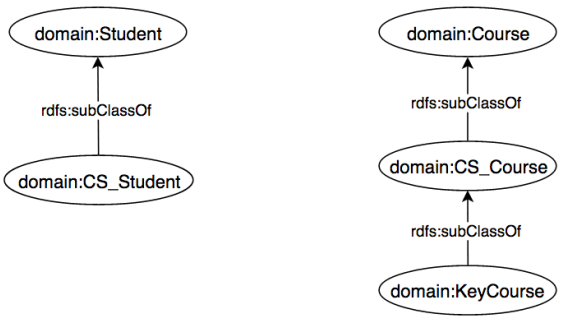

Figure 2.5: Ontology extension

In order 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 software for storage and retrieval of RDF triples. The retrieval, like by any other kind

of database, happens via queries, that 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 a 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 dataset from Figure 2.6, then it returns the *domain:1* and *domain:3* instances for the variable *?student*.

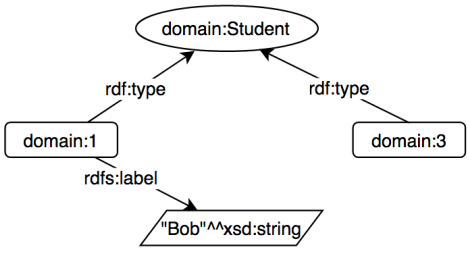

Figure 2.6: Example RDF dataset

Furthermore the triple pattern can consists 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*, that 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 sub pattern, 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 its 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 course 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. In order to connect these two, during the project our own 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 are 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.

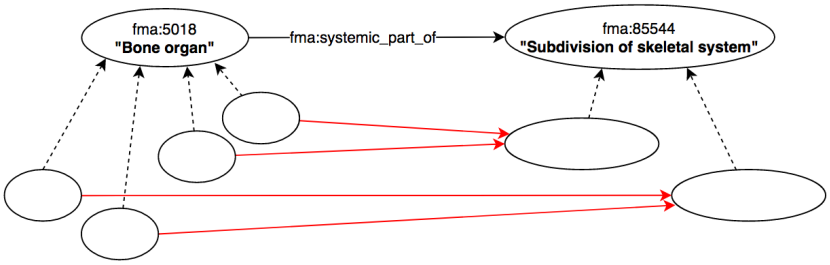

Figure 2.7: Ontology structure for the human skeleton

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 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) [1]. 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 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 an output. In our case input entities are segments of bone organs but they could be any kind of material. During a particular process the input entities are studied, and the result of the study is an in 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.

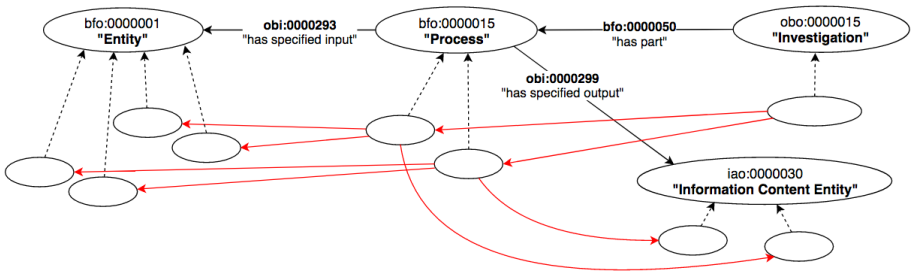

Figure 2.8: Extending OBI ontology

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 by means of 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 RDFBones ontology

RDFBones 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: RDFBones 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, *RDFBones* 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 ex-

tension, 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 RDFBones and thus the OBI to the FMA.

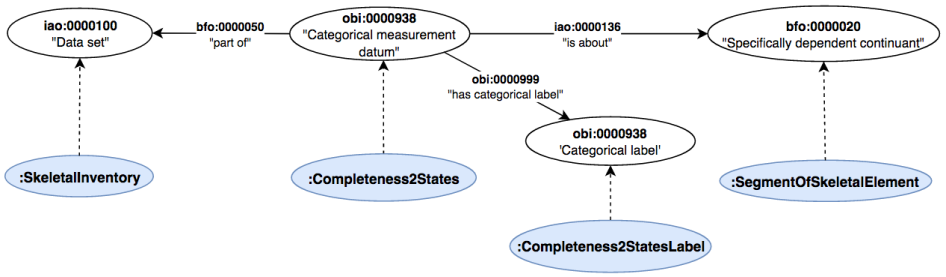

Figure 2.9: RDFBones as OBI extension

2.3 Web applications

Nowadays web applications are prevalent since they have the advantage against desktop application that they do not have to be installed on local computer, and can be accessed from 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. In order 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-sever architecture

Web applications consist of two main units. The first is client, which the user interacts with, and the second is the server, which is an 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.

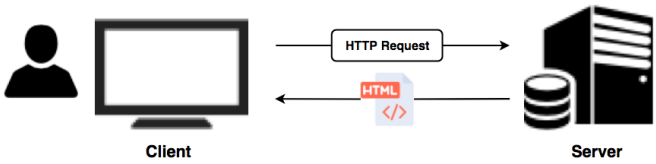

Figure 2.10: Client server communication

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 a 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 paramater (i.e *class* and *href*). The tag *div* is the most general element of web pages, while the *a* tag defines link, where the *href* parameter is defines the URI of the HTTP request they initiate. The request of the example link arrives to 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 to create, 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 a 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 a desired data. The result of a query in terms of the web application is conventionally a list of data objects. The term object is used generally, 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 basically 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 in 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, that 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 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 make the development more convenient and clear.

```
<#macro linkButton urlMapping id>
  <a href="webapp.com/${urlMapping}?id=${id}">
    <img src="webapp.com/images/jump.jpg"/>
  </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 an 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, in order to get data about individuals. Then the returned page may contain 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 by means of 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.

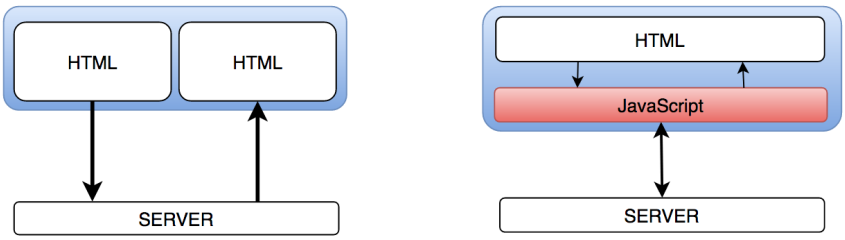

Figure 2.11: Static and dynamic web pages

Figure 2.11 illustrates the difference between static and dynamic web pages, where the blue rectangle stand for the client side. The idea of dynamic web pages in data driven web applications, that 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 id-s 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 is 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 as profile page, because it shows the data related to one individual RDF instance, and the latter will be referred as 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. By means of 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 the in the skeletal subdivision propery group the two faux properties (*skull* and *vertebral column*) can be seen. The instances (blue

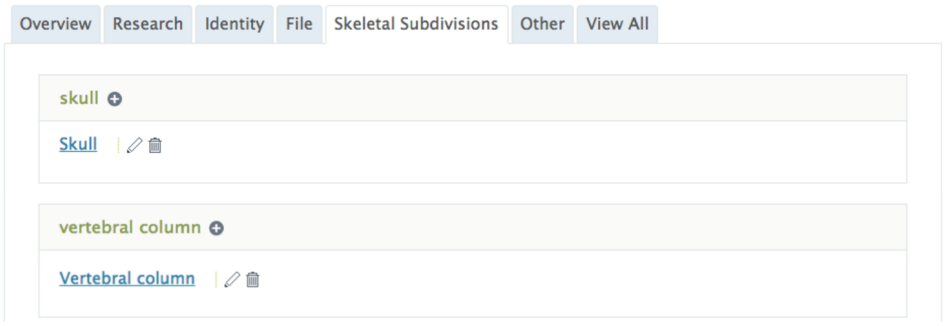

Figure 2.12: Faux properties on VIVO profile pages

entries) are both connected to the skeletal inventory instance with the *rdf-bones: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 propery are considered, the value for the custom entry form is for the data input.

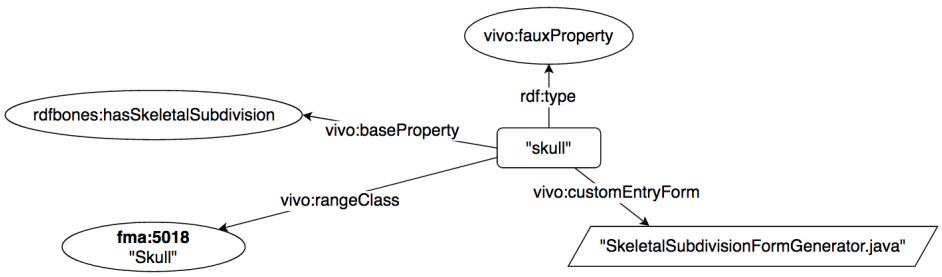

Figure 2.13: Faux property descriptor RDF dataset

On Figure 2.12 next to the property field, a button with a + sign can be seen, which is link to a data entry form page. From each property field the request arrives to the same handler routine on the server. These requests have always 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.

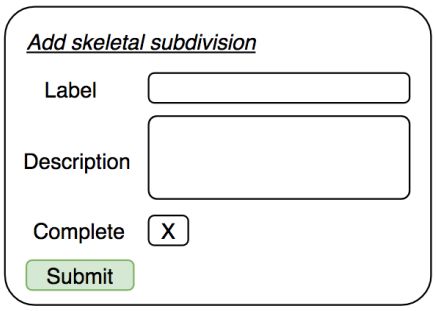

Figure 2.14: Custom entry form for skeletal subdivision

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 fields 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 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 defintion 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 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

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

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 and 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 sub parts, where these sub parts 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 enables to add the components

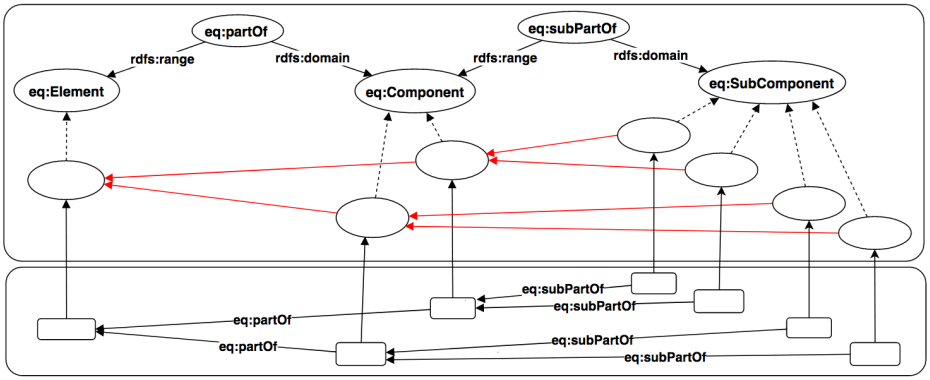

Figure 3.1: Ontology and RDF triples for complex entities

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

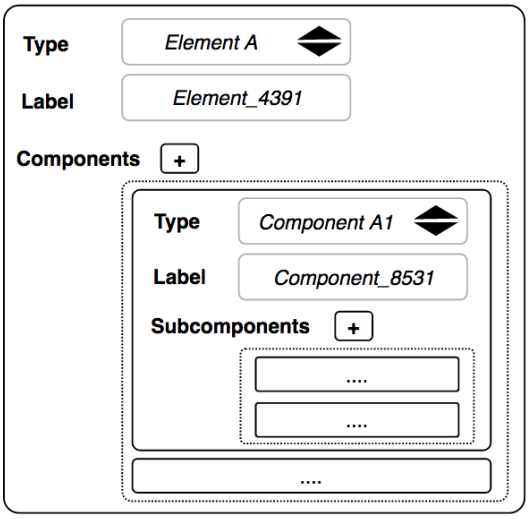

Figure 3.2: Multi level form

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 for 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

Further challenge for the client algorithm is that the options of the selectors for the component type on the sub forms are dependent on the selected type on the parent form (the form which the sub form 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 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 firstly 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 an other routine on the client has to reset the state of the form with the appropriate sub forms 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 sub forms, which is done through AJAX.

3.2 RDFBones use-cases in VIVO

In VIVO the data input problems were solved by means of a 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 *RDFBones* project.

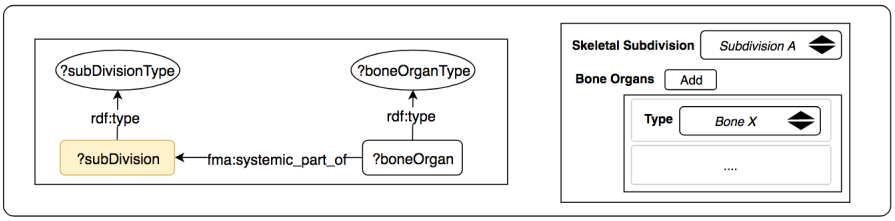

Figure 3.3: Skeletal subdivision graph pattern and form layout

The problem is that it would be necessary to define custom JavaScript routines for each case, which adds the sub forms and handles the dependencies between the particular selector fields, because the FTL library does 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 queries for the data retrieval.

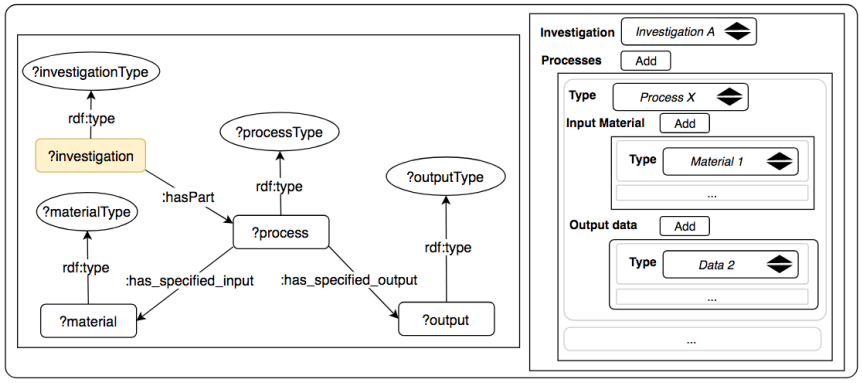

Figure 3.4: Investigation graph pattern and form layout

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 are 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 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, that 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 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 red arrow, and the multi triple with double line arrow.

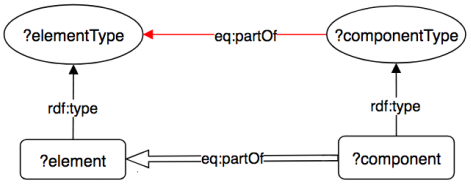

Figure 3.5: Extended RDF graph pattern definition

Above the data generation based on the graph pattern, 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.

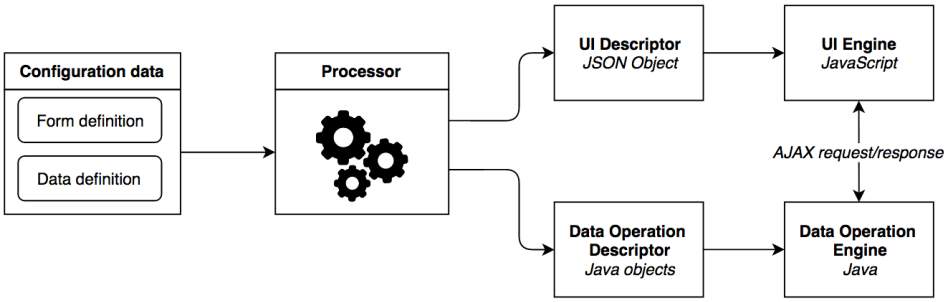

Figure 3.6: Framework functionality outline

Chapter 4

Vocabulary for RDF data input

The aim of this chapter is show how can different data input problems be expressed through the instantiation of 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.

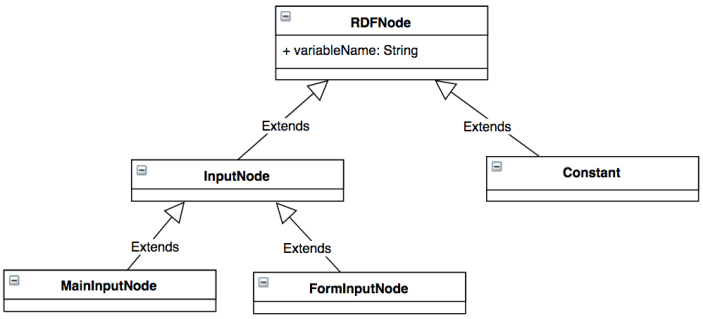

Figure 4.1: Java classes for RDF nodes

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.

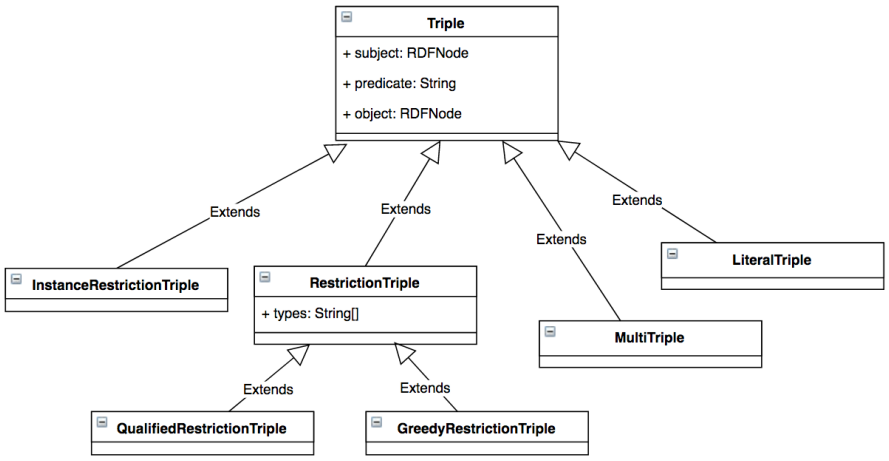

Figure 4.2: Java classes for RDF triples

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 *43*. 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*.

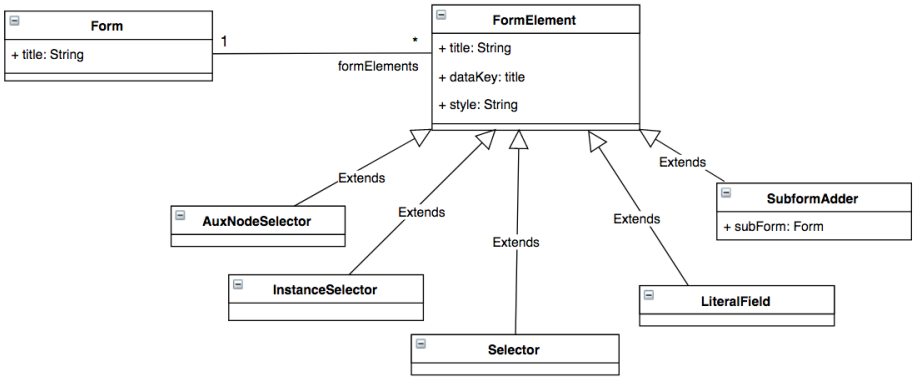

Figure 4.3: Java classes for the form

In the class *FormElement* the most important field is the dataKey, which is 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 to define 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 a such field where the existing instance can be selected through a floating window implemented in JavaScript. This window allows tp display more information, by displaying the instances within a table where the different columns contains 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.

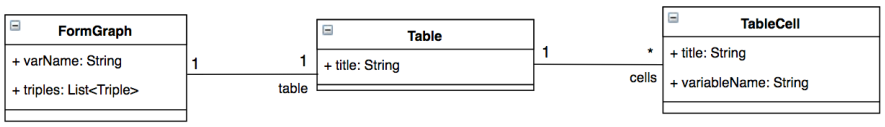

Figure 4.4: Java classes for instance viewer

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 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 by means of 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,

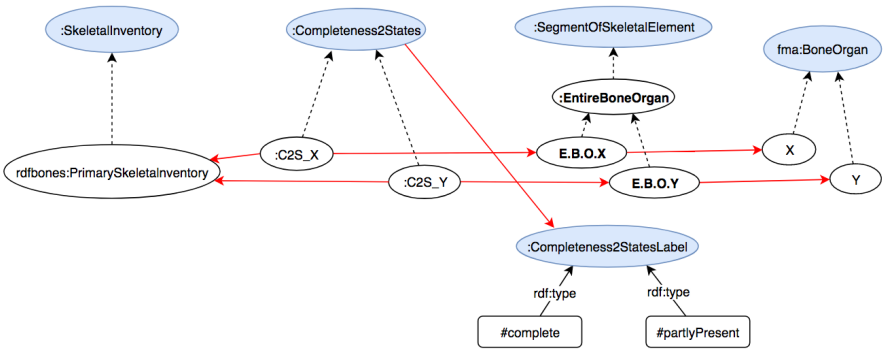

Figure 4.5: Primary skeletal inventory extension

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 two sub subdivisions, so a primary skeletal inventory for a skull is represented with the following triple pattern:

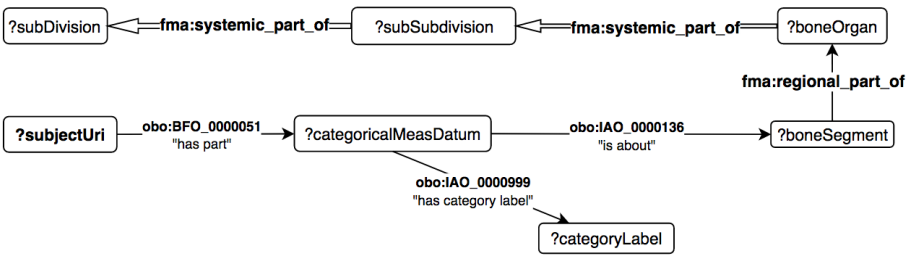

Figure 4.6: Skeletal inventory data triples

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

inventory. 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 sub form adder element appears. This element is implemented in a way that it incorporates a class selector as well. This is the same if the sub form would contain a type selector, just the form layout is more compact. It can be seen the added sub form has a title, which the label of the added class for the element. The sub form adder for the bone organs is the same. The sub form 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.

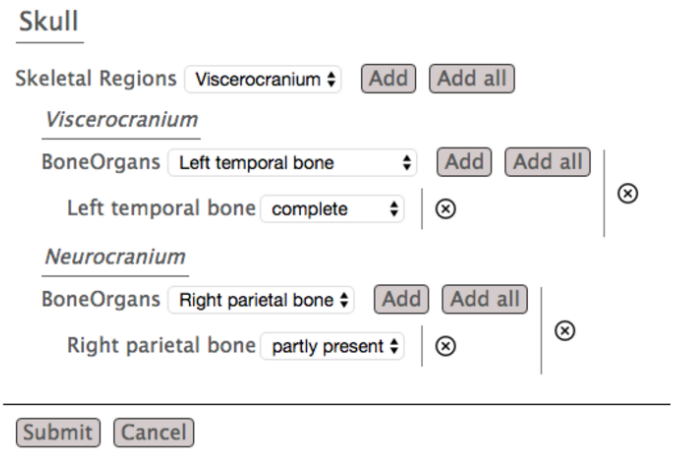

Figure 4.7: Generated interface for primary skeletal inventory

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 denotes 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 depicted with yellow are normal *RDFNode* instances in the descriptor dataset, but as they do not appear on the form, they will get a new unused IRI after the submission. The class

input nodes get their values based on the SPARQL queries generated from the restriction triples. The blue classes in turn are the classes that do not appear on the interface, but their value has to be evaluated based on value of the incoming bone organ type for the data creation. This shows that the interface can hide the complexity of the underlying data model from the user.

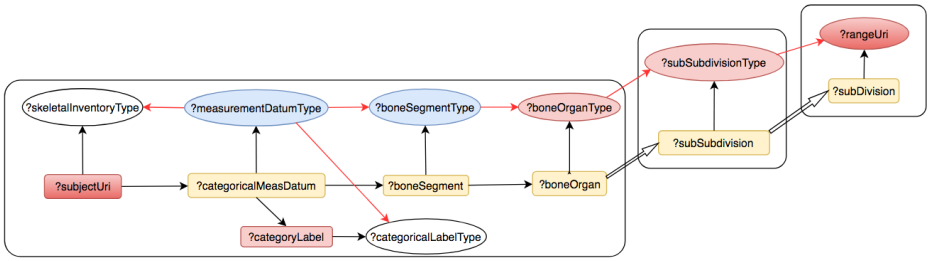

Figure 4.8: Complete data definition

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.

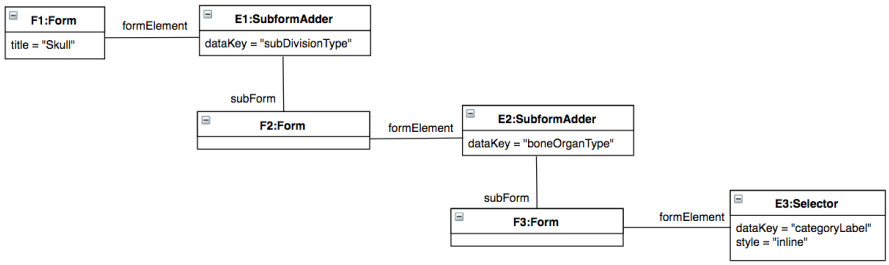

Figure 4.9: UML object diagram for form layout definition

4.2.2 Study Design Execution

The entry form for study design execution has as well the hierarchical layout like the one for skeletal inventories, but there are additionally two elements on the main form. The first is the selector from skeletal inventories. It plays a role by the selection of the bone organs as input for the assays. To each assay a set bone segment types is defined in the extensions, that can be can

be assigned to them as input. These bone on this form are not created newly but existing ones are selected, that were already added in the frame of the skeletal inventory data input. However there can be a large amount bone segments stored in the system, and thus the search is facilitated by showing only the ones that belong to the preselected skeletal inventory. The second is a global label field, whose value will be the label of all newly created instances.

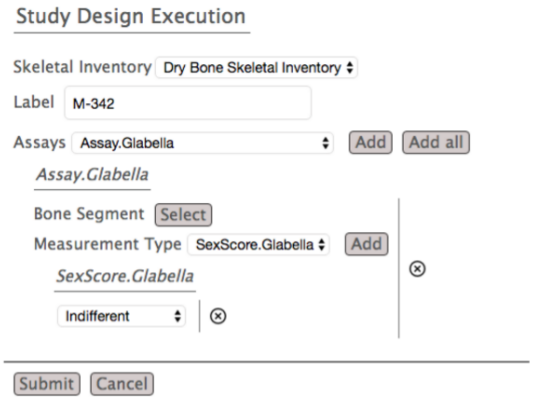

Figure 4.10: Input form for study design execution

Moreover the bone segment selector is not just a HTML selector input field, but a floating window implemented by JavaScript that allows the convenient browsing (Figure 4.11). It has two advantage with respect to the conventional selector. Firstly it allows to display additional information about the instances above their labels, like their types or longer descriptions. Secondly it does not loads the form layout with additional subform for the selected instances, which by large amount assays and measurement datums is an important aspect.

On Figure 4.11 can be seen that the there are two section, one for the selected instances, and one for the instances to select.

The complete data structure of the form can be seen on the following image.

- Validation - cyclic graph - multiple dependencies
- subgraph subform dependencies

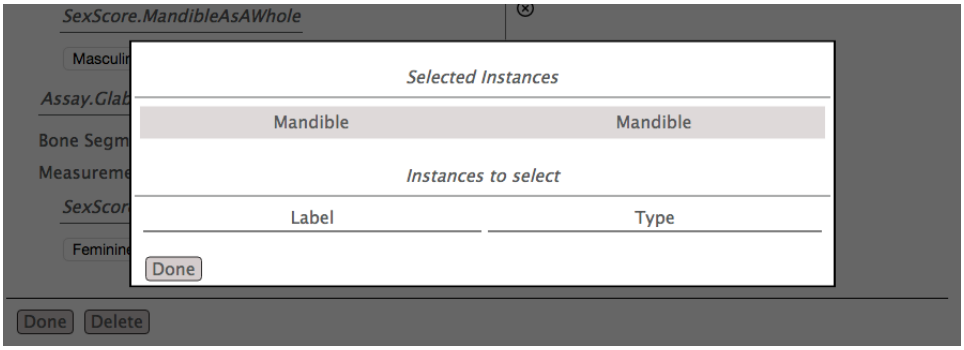

Figure 4.11: Instance selector for existing bone segment

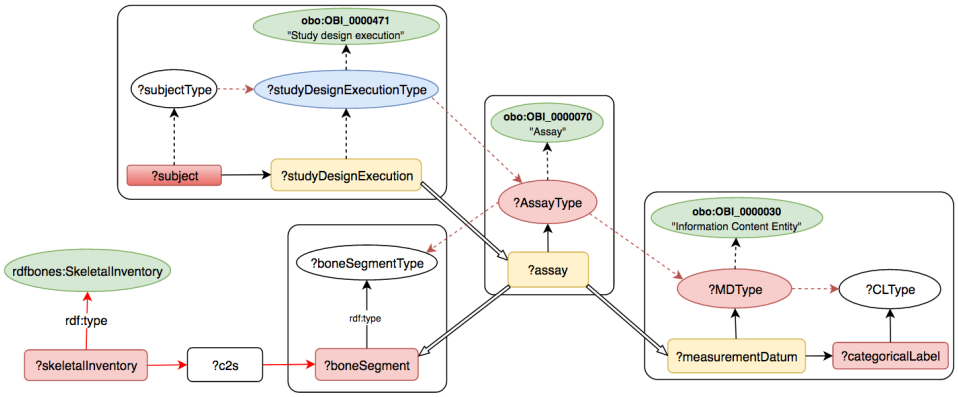

Figure 4.12: Complete data model

- Descriptor for data and processing

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 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 of the server and client side programming, including how the framework can be integrated into the applied web application. In both sections the explanation refers to the examples discussed in the previous chapter.

5.1 Main software modules and tasks

In Figure **??** we have seen the main work flow and components of the framework. Figure 5.1 is in turn a more detailed depiction of the software modules and processes of the application.

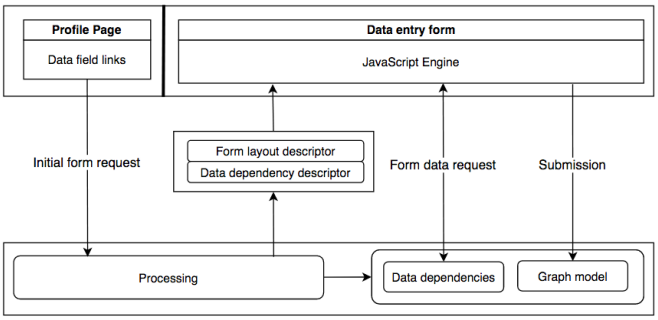

Figure 5.1: More detailed scheme

This section is divided into three subsections. First is the part (section 5.1.1) discusses the processing of the configuration data and generation of the functionality descriptor objects for the client and the server. 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 data is retrieved from the triples store.

5.1.1 Validation

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

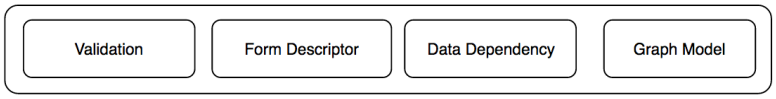

Figure 5.2: Processor tasks

The input of the algorithm is the set of triples describing the data model with its constraints, and 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 three scope of the checking, the nodes, the graph and the form.

Figure 5.3 depicts an example data model and illustrates the cases of valid and invalid nodes (Figure **??** contains the meaning of the shapes and colors). The explanation starts with the discussion of the form input nodes. Node *2* is valid because it is possible to generate a SPARQL query that retrieves the possible values of it. The query contains one triple which ask the subclasses of the constant class. Furthermore node *4* is valid as well, because there is path to it from a valid class node, therefore for there is again a SPARQL query for its values. However the variable *3* is not valid, because it does not contribute to any triple in 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 –> 1 –> 4 counts in the processor routine, but 2 –> 5 –> 6 –> 3 does not.

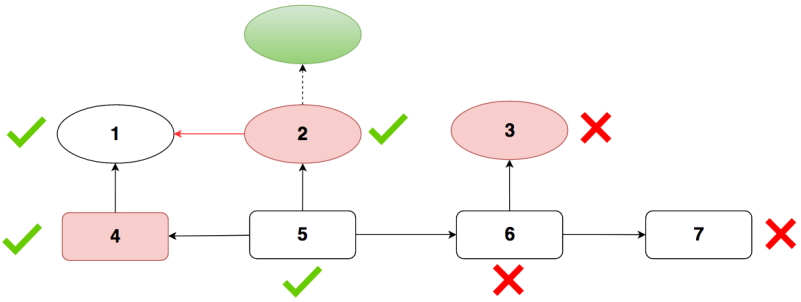

Figure 5.3: Valid and invalid nodes

The next task regarding the nodes is to check if each has a value 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 generated newly and get as value a new unused URI from the triple store, must constribute to triple as subject, where the predicate is *rdf:type* and the object is a valid class. For this reason the node *5* is valid, since its type class is valid, but node *6* is not. Moreover node *7* is not valid as well, since it does not have any type class defined in the data model. Finally regarding the literals the validation is the simplest, either they appear on the form or have constant value, otherwise they are invalid.

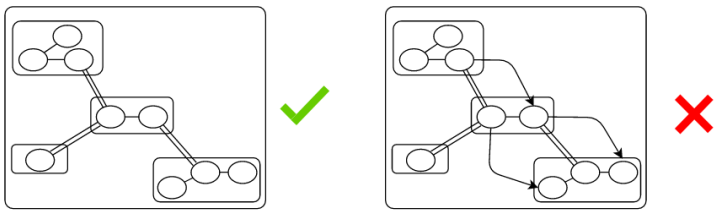

Figure 5.4: Valid and invalid graph

Above the nodes important itself the whole graph built by the triples have to be investigated. In the previous chapter the triple type *MultiTriple* were introduced. The rule regarding this type of triple that it divides the graph into subgraphs, and the subgraphs can connect to each other only be these triples. Figure 5.4 illustrates the valid and invalid graph arrangements.

The reason is that only to this type of scheme can the JSON object of the created by the data input process be mapped.

5.1.2 Dependencies and form functionality

The set of triples describing the data model builds a graph where the various input nodes are connected to each other. Further task of the processor algorithm to determine the subgraphs that define the SPARQL queries for the node appear on the form. The elements on the form has a specified order, and the rule that have to be considered during the processing, is that a variable can be dependent only of the main input variables or such variables that are before it on the form. The reason is that the dependency is practically SPARQL query with one output and with one or more inputs, and input node have to available by the execution of the query.

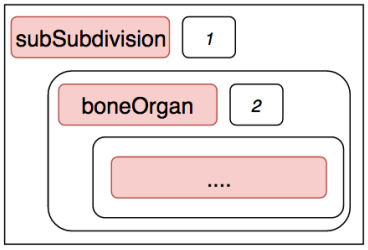

Figure 5.5: Form element order

Figure 5.5 shows the simplified form layout of the skeletal inventories and emphasizes the order of the elements with the number. The idea is that values of the selector for the node *subSubdivision* can be only dependent on the main input nodes because it is the first node. The *boneOrgan* in turn can be dependent on the *subSubDivision* too because its value has to be set before the options of the subform is loaded, thus it can be substituted into a SPARQL query.

Figure 5.6 depicts the scheme of the data constraints of the skeletal inventory, based on which the dependency retriever algorithm can be exemplified. As it was mentioned the task is to get one or more path from the variable of to an other node whose value is available for it. As the form's first element refers to the node *subDivision* its dependency has to be evaluted first. Since the node *subSubDivision* contributes to two restriction triples in the data

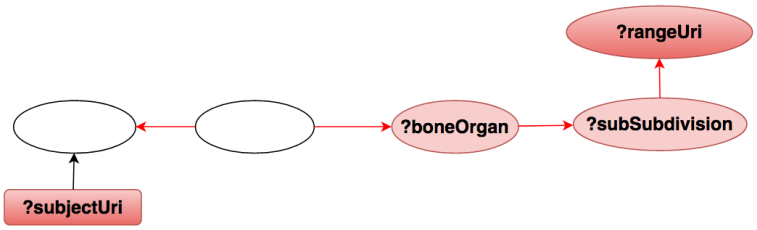

Figure 5.6: Skeletal inventory data constraints

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*). While the by dependency for the node *boneOrgan* the *rangeUri* does not count, since the algorithm terminates already by the *subSubdivison* variable. Figure 5.7 shows the results subgraphs of the algrithm, where the output is depicted with green and the inputs with red and light red respectively.

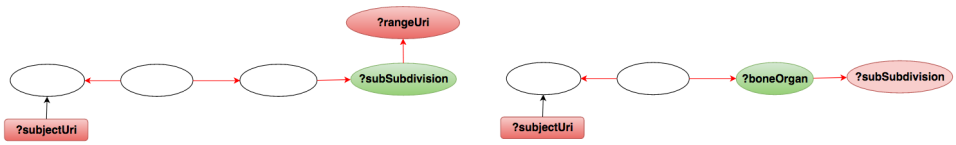

Figure 5.7: Form dependency subgraphs

This result of the processor concerns both the descriptor of the client, and the server. On the server the depencies are stored a classes that have the necessary fields, for the inputs and output and for the triples. Then these initialized dependency descriptor class objects are stored in a data map, where the key is the variable name of the output node. This map is a field of the form configuration class, thus if the client asks for the appropriate form variable through AJAX these SPARQL query assemble by the triples for can be executed with the incoming values.

For the client the dependency description is just the an assignment of an array, which contains the input node of the dependency, to the form node. In the case of the *subSubdivision* it is an empty array, because it is not dependent on any form element, but for example the *boneOrgan* in the use-case for the study design execution the is dependent from the *assayType* and the skeletal inventory. The task of the form by loading new sub form is

to check this array and get the variables required values from the form. This mechanism will be discussed in more detail in section **??**.

Above the dependency the descriptor the JavaScript framework obtains the descriptor data file for the form elements upon which it can generate the form. The mechanism of the form description retrieval a quite simple because it is practically the convestion of a Java object into a JSON object. The fields of the Java object appear in the JSON objects as key. In

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

Listing 5.1: Java to JSON

Where the *this.title* has contains the value for the title of the form object in the descriptor Java object. The same way if the Java class has list field, like the form has list of form elements, then getDescriptor() routines of each element is called and inserted into a JSON array. Moreover if the form element is sub form adder then it has a field *subForm*. This comes as well of course into the descriptor, and this is the way how the multi level JSON is created.

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

Listing 5.2: Subform descriptor

Figure 5.8 illustrates the generated JSON object for a form with the data dependencies too. The task of the JavaScript routine to interpret this configuration data and generate the form and subform upon user action.

5.1.3 Graph model generation

The previous section outlined how the set of Java objects are converted into the form decriptor JSON object, and into Java objects describing the variable dependencies for the AJAX requests. Further task of the framework

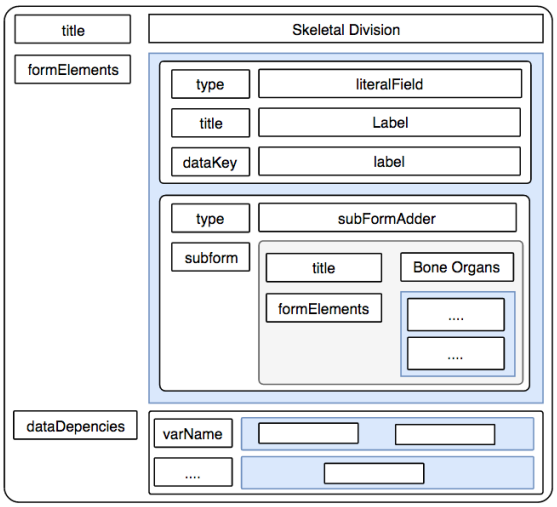

Figure 5.8: Form descriptor JSON

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 chapters, that the form data object has the same scheme as the form has, and 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*.

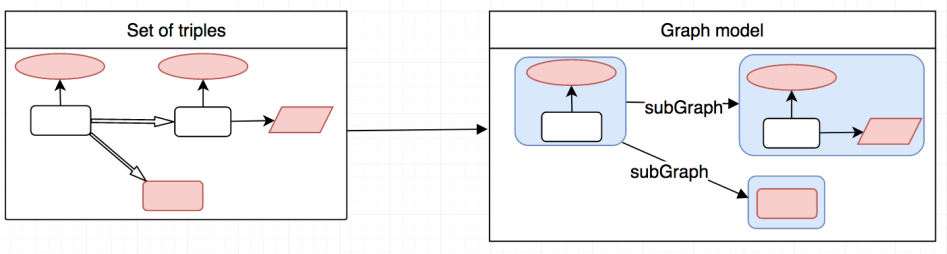

Figure 5.9: Conversion from triples into graph model

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 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 5.3: Subform descriptor

The save routine creates the RDF triples of the graph based on the data fields of the JSON object. The a 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 retriaval, where the graph performs the SPARQL queries based on its triples, and generates the arrays of objects from the result table.

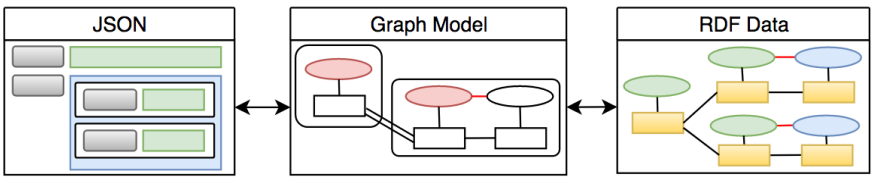

Figure 5.10: Graph decomposition

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 properly the functionality of the server. While in the last subsection it is discussed how can the developed framework integrated into the VIVO web application.

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 sub form 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 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 betwen these JavaScript classes to Java classes, that they do not contain only fields and routines (or methods) but UI elements as well. The UI elements can represent and 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 processes starts 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

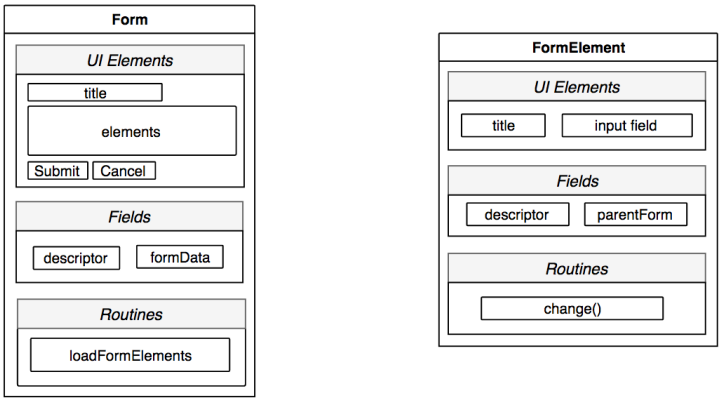

Figure 5.11: Form and form element classes

the descriptor contains a list of the form element descriptor objects. The *loadFormElements()* routine iterates through on this array, and initiates the form element objects.

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

Listing 5.4: 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, that contains their title and input field HTML element. This container field is added to the *elements* field of the *Form* object.

Listing 5.5 show a small cut from the code of the StringField, 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 StringField 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 5.5: Form element

This is the basic mechanism of how object-oriented JavaScript can be employed to generate forms, and put the entered values in to 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.

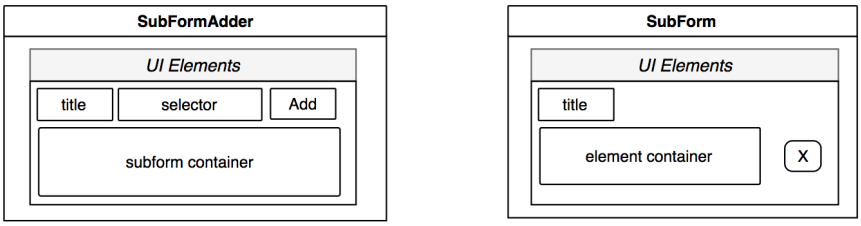

Figure 5.12: SubForm and sub form adder

Figure **??** 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 sub forms, which are appended into the sub form 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 5.10 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 sub form data object will be added dynamically upon the click events. So if the user clicks the add button, then new object is initialized (*subFormDataObject*), which will be the data object of the sub form. Important to note that this object will contain value of the selected option of the sub form 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 sub form container of the sub form adder.

```
class SubformAdder {

  constructor(descriptor, formData){
    this.addButton = $("<div/>").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 5.6: 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 covers the functionality of the server. On the client side the implemented JavaScript classes were just included into the form template file, but the framework integration to the server is a more complicated issue, therefore the first subsection is dedicated to it. Furthermore the implementation of the form data calls with the variable dependency calls are discussed here, as well as the routines how the graph model can save, edit and retrieve the or subsets of the form data.

VIVO integration

As it was mention in the previous chapter the process of the form loading start with the profile pages of VIVO. In VIVO the entry form loading is initiated by the property fields. Normally the programming in VIVO happens through so called generator classes. The task of the generator classes is to define the dataset that have to be created in the form, and reference the Freemarker template file for entry form.

```
<someProperty>  vivo:customEntryFormAnnot
                    "rdfbones.DrawingAConclusionGenerator.java"
```

Listing 5.7: Custom entry form definition in VIVO

The approach implemented in the frame does not replace this structure, but makes it simpler. In our cases we use as well generator classes, but the definition of the data happens through the intialization of an instance of the class *FormConfiguration* instance (listing **??**).

Figure 5.13 shows the process of loading an 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 executed, and the necessary JSON and Java object are generated. Afterwards the server saves the form configuration object its cache with a key, which is called in VIVO *editKey*. The box in the middle of the image shows, that the response web page includes the JS library (simplified notation *framework.js*), and the value of the *edit*. This is the value will be sent in each AJAX request to the server, and based on this the can the server

find the form configuration instance that returns the JSON object for the client.

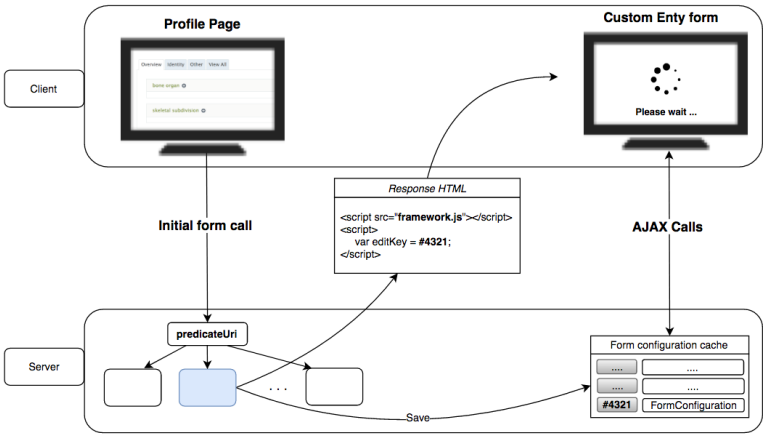

Figure 5.13: Form loading process

Figure 5.14 show the UML class diagram *FormConfiguration* class. The fields of the classes were already discussed in the previous chapter, but here it is important to note that each AJAX request is server by the method *serveRequest()*, where both the input and the output is JSON object.

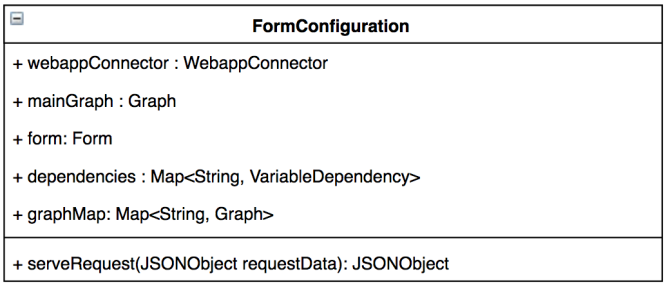

Figure 5.14: UML class diagram for FormConfiguration

Listing 5.11 shows the main scheme of *serveRequest()* routine. Each request coming from the client has the key *task*, which defines what data JSON object has to be served for the client.

```
SONObject serveRequest(JSONObject requestData){
    switch (requestData.get("task")) {
      case "formSubmission":  ... break;
```

```
        case "formDescriptor":   ... break;
        case "editData":     ... break;
        case "deleteAll":    ... break;
    }
}
```

Listing 5.8: AJAX request server routine

Finally a really important part of the integration is the access to the used triple store. For this purpose an interface, called *WebappConnector* is defined. It defines the functions that allows the querying and manipulation of the triples.

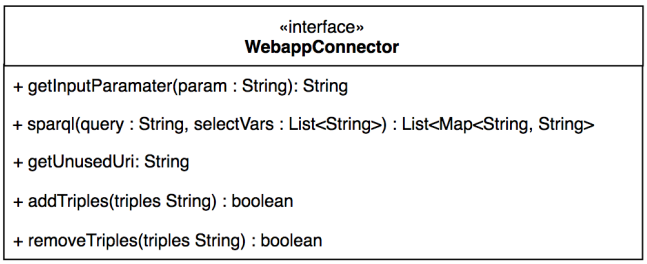

Figure 5.15: UML class diagram for WebappConnector

Form data loading

Important part of the framework functionality is the loading of the dependent data to the forms. This is defined through the keyword *formData* in the task parameter. Moreover it has parameter, the *variableToGet*. Based on this variable, the form configuration finds the variable dependency instance and passes the incoming JSON object to its method *getData()*.

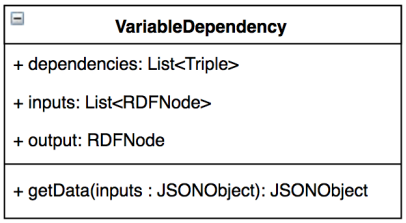

Figure 5.16: UML class diagram for VariableDependency

```
JSONObject serveRequest(JSONObject requestData){
switch (requestData.get("task")) {
  ...
  case "formData":
    return this.dependencies.get(requestData.get("varToGet")).
        getData(requestData)
  ...
}
```

Listing 5.9: Loading form data from FormConfiguration

We have seen that the dependency is set of triples, which build a graph. This graph can be built by class restriction triples, which constitutes to the generated SPARQL query not with one triple but with three triples.

```
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 5.10: SPARQL query generated by class restriction triple

Saving, editing and retrieval of RDF Data

```
JSONObject serveRequest(JSONObject requestData){
switch (requestData.get("task")) {
...
case "saveData":
return this.mainGraph.saveData(requestData.get("formData"))
case "retrievedData":
return this.mainGraph.getData(requestData.get("startUri"))
case "editData":
return this.graphMap.get(requestData.get("varToEdit")))
.editData(requestData)
...
}
```

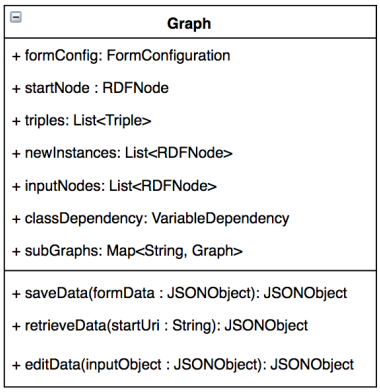

Figure 5.17: UML class diagram for Graph

Listing 5.11: Loading form data from FormConfiguration

Chapter 6

Conclusion

6.1 Evaluation

The three main tasks of a web application for data management is to allow the user to create, edit and delete particular set of the data. To achieve this several interface elements are required on the web page with the user interacts with. The data manipulation is initiated always by user actions, which trigger different request from the client and user. The task of the server is to interpret the incoming request and perform the necessary data operation in the database. The most important utility of the implemented framework that it extends the VIVO framework so that more complex data input problems can be defined only by the declaration of the data scheme and the layout of the user interface, and the developer does not have to care about how the client and server routines realizing the data flow between the user and the database. It is achieved such a generic interface routine, which offers the defined input elements and automatically handles the add, edit and delete operations, while the server can interpret the request and based on the data definition is capable of creating, editing and retrieving the data in scope.

Furthermore by such RDF data based systems, that every entity of the domain is represented by a class of the ontology and how the individual entities are related to each other is in turn defined in OWL restrictions. This means in terms of interface of a web application for creating RDF data, that its elements are dependent on the ontological statements. Thus according

to type of the entity about the data has to be created, the corresponding classes from the ontology have to be retrieved, and loaded to data input elements. In our data scheme descriptor vocabulary it is important the these restrictions can be addressed, and the framework can interpret it. So the implemented framework based on the scheme definition is capable of query the ontological statement and generate the interface. This allows the usage of the framework for other kind of processes where the rules of the domain is described by ontology extensions.

This approach can significantly accelerate the development and allow the Semantic Web based application development for individuals without experience with web application and programming technologies.

6.2 Future work

Appendix A

Glossary

Just comment `\input{AppendixA-Glossary.tex}` in Masterthesis.tex if you don't need it!

Symbols

$	US. dollars.

A

A	Meaning of A.

B

C

D

E

F

G

H

I

J

M

N

P

Q

R

S

T

U

V

W

X

Appendix B

Appendix

B.1 Something you need in the appendix

Just comment `\input{AppendixB.tex}` in Masterthesis.tex if you don't need it!

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] Anita Bandrowski, Ryan Brinkman, Mathias Brochhausen, Matthew H. Brush, Bill Bug, Marcus C. Chibucos, Kevin Clancy, Mélanie Courtot, 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.

[2] 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/.

[3] 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/.

[4] Felix Engel, Stefan Schlager, and Ursula Witwer-Backofen. An infrastructure for digital standardisation in physical anthropology. 11, 04 2016.

[5] 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/.

[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.