XML-Schema for Query Answering Interface in DIG 2.0

Santy Sari Sugianto
Matriculation Number 31661
Student Project

Supervised by
Prof. Dr. Ralf Möller
Alissa Kaplunova

Software, Technology and Systems (STS)
Hamburg University of Technology
January 2007

Contents

ACKNOWLEDGEMENTS

I take this opportunity to thank all those persons who rendered their full services to my project work.

First of all, I would like to thank Prof. Dr. Ralf Möller from Software, Technology, and Systems (STS), Technical University of Hamburg-Harburg, for providing this topic of Student Project and thus, giving me a chance to further discover about Description Logics, DIG Interface, RacerManager, and go deeply into XML Schema, XSLT and XPath.

I would like also to express gratitude to my supervisor Alissa Kaplunova, for timely and kind help, guidance and valuable suggestions, and giving me the idea and suggestion to solve the problem that I encounter during the process work. And not forgetting those great moments where she always encourage me. This inspiration really help me to accomplish my project on time.

I would like also to thank the almighty God for His blessing and guidance during the making of this project work.

Chapter 1

Introduction

1.1 Background

World Wide Web (WWW) is a medium that was designed to be read by people. It means that a computer can not accomplish the tasks without human direction. The Semantic Web was developed to overcome this issue. A Semantic Web is a vision of web pages which allowed machines to process and perform actions automatically. To represent the knowledge based on formal semantics, we need some logics, one of them is Description Logics (DL) [11]. DL is very useful for defining, integrating, and maintaining ontologies, which provide the Semantic Web with a common understanding of the basic semantic concepts used to annotate Web pages.

Description Logic reasoners are needed to process Knowledge Base(s) (KBs) and make the implicit information explicit. There are many reasoners available on the market, each of them has their own interface. The Description Logic Implementation Group, known as DIG [7], proposed a specification of a standardized interface for DL reasoners.

This Description Logic Interface (also known as DIG) provides uniform access to Description Logic reasoners. The interface defines a simple protocol (based on HTTP PUT/GET) along with XML Schema that describes a concept language and accompanying operations.

The DIG interface is not intended as a heavyweight specification of a reasoning service. Rather, it provides a minimal set of operations (e.g., satisfiability and subsumption checking and classification reasoning) that have been shown to be useful in applications.

The proposed DIG version that is in our subject is DIG 2.0 [16]. The older version, DIG 1.1 has some limitations. Its expressivity is not sufficient to capture general OWL-DL ontologies - in particular datatype support is lacking in DIG 1.1 and there is a poor fit between DIG's notion of relations and OWL properties.

DIG 2.0 draws on experiences gained from earlier DIG specifications (1.0 and 1.1). However, the version 2.0 is not intended to be backwards compatible with existing DIG 1.1 implementations.

DIG 2.0 uses the OWL 1.1 (The Web Ontology Language) specification [8] for the definition of the language describing ontologies. It means, that any DIG 2.0 implementation is guaranteed to be compatible with OWL 1.1. Furthermore, this is expected to reduce the burden on the implementors of DL reasoners and ontology management systems.

Querying in DL is not the same as querying in database. The way DL systems process the query and the result of query answering are different from database query processing. The database query will return the result that explicitly available in the database. In the contrary if reasoning involved, we can get not only explicit data but also the implicit one. In order to query DL-based systems, we need a standard query language. Query Answering Interface is proposed to be used as a part of DIG 2.0. The previous versions of DIG already support query answering in a simple manner. E.g., we can ask for all individuals of the given concept or whether the concept is satisfiable. But these querying facilities are not expressive enough for modern DL reasoners.

Since the Query Answering Interface as part of DIG 2.0 is intended to become standard, we want to make it compatible also with other standard query languages, e.g., with OWL-QL. To achieve that, we propose XML

based translation rules from DIG 2.0 to OWL-QL and vice versa. These rules intended to be used by servers that support OWL-QL in order to communicate with DL reasoners that support DIG 2.0.

1.2 DIG Interface

DIG provides a basic API to a DL system that can be adopted by any parties. The interface that offered by DIG is relatively lightweight and provides just enough basic functionality to allow tools such as ontology editors to make use of a DL reasoner. DIG provides primitives for manipulation of DL ontologies, such as asserting and retracting axioms, as well as primitives for asking questions about ontology entities.

The core DIG specification is an XML Schema [9] for a DL concept language, ask/tell functionality and a description of a protocol used to communicate these operations. At the moment, DIG uses HTTP as its underlying protocol for communicating with a reasoner. The reasoner accepts HTTP POST/GET requests and responds as appropriate.

Any applications making use of DL reasoning may benefit from DIG. DIG does not intend to provide what we might truly call a reasoning service, but rather helps to insulate applications from the location and implementation language of a DL reasoner. The specification does not address issues such as stateful connections, transactions, concurrency, multiple clients and so on. That is why it is expected to be part of the larger architecture.

1.2.1 DIG 2.0

DIG 2.0 [6] is specified by means of class diagrams expressed in the Unified Modeling Language (UML) . This style allows for an unambiguous specification of the API primitives at a conceptual level. The conceptual aspects of API primitives, such as the types of arguments and the semantics, are thus decoupled from a concrete protocol and syntax used to actually access DIG 2.0 implementations. Hence, the same primitives can be realized using

different access protocols.

DIG 2.0 is a client-server protocol. The protocol is specified in two parts: the first part defines the abstract protocol, and the second part defines the binding of the protocol into a concrete transport mechanism.

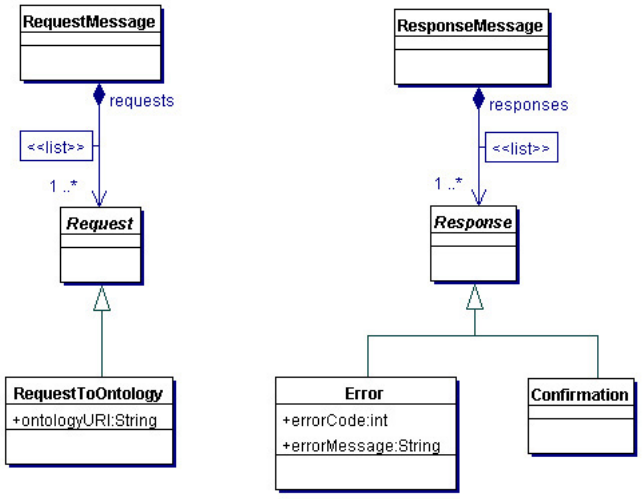

Figure 1.1: UML Diagram of Request-Response

The basic interaction pattern of this protocol (The correspondence class diagram is shown in Figure 1.1) is realized as request-response message:

- Each request is defined by its type and the (possibly empty) set of parameters. To request a service from a DIG 2.0 server, a client constructs a request of the appropriate type and sends it to the server.

- After receiving a request and processing it, a DIG 2.0 server constructs an appropriate response object and sends it to the client. Depending on the type of the request, the server should select the appropriate subclass of the Response UML class.

Each request is paired with exactly one response. The way in which requests are matched with their respective responses depends on the binding of the DIG protocol into a transport mechanism. A way to match requests

with responses is to use messages, which means each request message must correspond to exactly one response message.

A DIG reasoner may be used to implement services that provide concurrent access, transactions etc. Such functionality is not inherently supported by DIG 2.0.

1.2.2 DIG Extensions

Different Description Logic reasoners implement different DL sublanguages and different reasoning facilities. To partly support these differences, DIG 2.0 provides an extensibility mechanism. Implementations that offer functionality beyond this specification are free to provide their own primitives and use them simultaneously with the core DIG 2.0 primitives.

Each DL Reasoner has the freedom to extend the base interface schema that offered by DIG to adjust with their own logical environment. Technically an extension consists of two documents, namely an XML Schema document defining the syntax of messages and an associated HTML document providing the remainder of the syntax and sufficient information about the meaning of these messages for support implementation and usage of the extension. The first specifies the offered service syntactically. The latter should, at a minimum, describe the service results as well as error messages of the extension. Both documents should be available on the Web, and, by convention, their URIs must differ only in their extension (i.e. XSD resp. HTML extension).

The DL Implementation Group has developed a selection of extensions, which provide interfaces to some functionality the DL community. Some of those extensions are:

- Accessing Told Information [17]: In many applications (for example for debugging a knowledge base created by several clients) it is useful to be able to access the unprocessed information sent to a DL reasoner. To this end, a DIG 2.0 extensions provides the ability to retrieve the

information that has been explicitly given to the reasoner ("told") as axioms.

- Non Standard Inferences [18]: NSIs are increasingly recognized as a useful means to realize applications. For example, Least Common Subsumer provides a concept description that subsumes all input concepts and is the least specific (w.r.t. subsumption) to do so. A DIG 2.0 provides a proposal for an extension supporting a particular set of NSIs.

- Concrete Domain Interface [12]: In many practical applications, reasoning over specific domains with fixed (concrete) semantics such as integers, reals and strings is required. For this propose, modern DL reasoners provide for a so-called concrete domain support, such that constraints over values from concrete domains referred to by multiple individuals can be postulated (e.g., linear inequations over polynomials or equations over strings).

- Query Answering Interface [3]: We will describe this extension further more in the next chapter.

1.3 Structure of Work

The main objective of this student project is to develop an XML Schema as a frame rules for Query Answering Interface in DIG 2.0, and to do the translation from OWL-QL to DIG 2.0 and vice versa. So we divided this work into 4 chapters.

In the next chapter, we will discuss about XML Schema for Query Answering Interface. It consists of the description of the XML Schema in general, syntax of Query Answering Interface, and the XML Schema for Query Answering Interface.

Chapter 3 describes how to do the translation from OWL-QL to DIG 2.0 and vice versa using XSLT. It also discusses about the reason using XSLT in this project, and mentions RacerManager as one of the OWL-QL server

which can be used in future as reference implementation using our translation schema.

The last chapter presents conclusion of this student project as well as gives the summary and suggestions for possible future work.

Chapter 2

Query Answering Schema

2.1 Motivation

There are some expressive and decidable query languages supported by modern DL reasoners (like RacerPro [2]), e.g., nRQL [1] which is a native query language of RacerPro. But these query languages are not standardized. Since the DIG 2.0 intended to provide a standardized access interface for DL reasoners, it needs a standardized query language. Therefore, Query Answering Interface is proposed to be used as a standard query language. In order to validate XML documents containing queries and answers, we need to define a Query Answering XML Schema.

2.2 XML Schema

The purpose of the XML Schema is to define a class of XML documents, and so the term "instance document" is often used to describe an XML document that conforms to a particular schema. XML Schemas express shared vocabularies and allow machines to carry out rules made by people. They provide a means for defining the structure, content and semantics of XML documents. This definition includes what elements are (and are not) allowed at any point; what the attributes for any element may be; the number of

occurrences of elements; etc.

The majority of XML documents are "well formed" rather than "valid". The former means that there is exactly one root element, and every sub-element (and recursive sub-elements) have delimiting start- and end-tags, and that they are properly nested within each other. A document is called valid if it is "well-formed" and conforms to a specified set of production rules.

To validate an XML document, some form of validating rules need to be provided. This can be done by any Document Type Declaration (DTD). An XML Schema sounds look like a DTD, however there are some critical differences. The most notable being that XML Schema can deal with namespaces, and DTD's can not. Moreover, XML Schema can mix namespaces. It must be mentioned that XML Schema is very dependent on namespace.

An XML namespace is a collection of names, identified by a URI reference, which are used in XML documents as element types and attribute names. The namespace does not need to refer to a valid location. Namespace support in XML Schema is flexible yet straightforward. It is not only allows the use of any prefix in instance documents but also lets us open our schemas to accept unknown elements and attributes from known or unknown namespaces. Each XML Schema document is bound to a specific namespace through the *targetNamespace* attribute, or to the absence of namespace through the lack of such an attribute. We need at least one schema document per namespace we want to define (elements and attributes without namespace can be defined in any schema, though).

2.2.1 XML Schema Types

There are two types of XML Schema:

- Simple Type : Elements that do not have other subelements are categorized as this type. Attributes are also included into this type. In the example below, we define the tag *predicate* taken from Query Answering proposal schema to be of simple type because it is intended to not have subelements.

```
<xs:element name="predicate" type="Variable"/>
```

- Complex Type : Elements that contain other subelements or attributes are categorized as this type. The following example taken from QueryAnswering schema shows that*PredicateQueryAtom* categorized as complex type because it has subelements (e.g., *ConcreteDomainQueryVariable*, *predicate*, etc)

```
<xs:element name="PredicateQueryAtom">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref ="ConcreteDomainQueryVariable"/>
         <xs:element ref ="owl11xml:Constant"/>
         <xs:element ref="predicate" minOccurs="0"
          maxOccurs="unbounded"/>
         <xs:element ref="lambda" minOccurs="0"
          maxOccurs="unbounded"/>
         <xs:element ref="op" minOccurs="0"
          maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Besides those two types, XML Schema also provides *group* tag that allows for grouping elements, and *attributegroup* tag for grouping attributes. Using these tags the readability of schemas can be improved and updating schemas can be facilitated easier. In fact, a group can be defined and edited in one place and referenced in multiple definitions and declarations.

In the following example, we declared a group *BooleanConstructGroup* consisting of three elements of choices, that are *QueryObjectIntersectionOf*, *QueryObjectUnionOf*, and *QueryObjectComplementOf*.

```
<xs:group name="BooleanConstructGroup">
   <xs:choice>
      <xs:element name="QueryObjectIntersectionOf"/>
```

```
      <xs:element name="QueryObjectUnionOf"/>
      <xs:element name="QueryObjectComplementOf"/>
   </xs:choice>
</xs:group>
```

2.2.2 Occurrence Constraints

The occurrence constraints in XML Schema are defined by *minOccurs* for minimal occurrence and *maxOccurs* for maximal occurrence of elements or attributes. The default value for both the *minOccurs* and the *maxOccurs* attributes is 1. Thus, when an element is declared without a *maxOccurs* attribute, the element may not occur more than once. Similarly, when the element is declared without a *minOccurs* attribute, the element must occur at least 1. If both attributes are omitted, the element must appear exactly once.

```
<xs:element name="SupportedRequest" minOccurs="0" maxOccurs="unbounded">
```

The example shows that the element *SupportedRequest* is optional and can be used unrestricted times.

2.2.3 Global Groups, Elements, and Attributes

Global groups, global elements, and global attributes are created by declarations that appear as the children of the schema element. Once declared, they can be referenced in one or more declarations using the *ref* attribute as described below. In the following example, we refer to the group *BooleanConstructGroup* mentioned before (see Section 2.2.1).

```
<xs:complexType name ="QueryBodyType">
   <xs:sequence>
      <xs:group ref="BooleanConstructGroup" minOccurs="0"
       maxOccurs="unbounded"/>
      <xs:group ref="QueryAtomsGroup" maxOccurs="unbounded"/>
   </xs:sequence>
</xs:complexType>
```

2.3 Query Answering for DIG 2.0

Query Answering schema is divided in two parts, the Request part and the Response part. In the Request part, the syntax of the requests send from clients to DL reasoners is defined. The tell language is also a part of the request. In the Response part, the syntax of the answers send from DL reasoners to clients is described. In the next subsections, we will discuss more deeply the mentioned main parts of the Query Answering Interface, in particular, *DIGDescription*, *Retrieve*, and *QueryAnswers* constructs.

2.3.1 DIGDescription

DIGDescription response is an answer from the DL server as a reply to *DIGDescribe* request. One of elements the *DIGDescription* constists of is *supportedRequest*. It describes which functionality is supported by the DL reasoner, such as *Retrieve* for query answering procedure.
Below is the example of the *DIGDescription* construct.

```
<DIGDescription
      name="Racer"
      version="1.9.5"
      message="Racer running on localhost"
      supportsLanguage="SHIQ(D)"
      supportsAnnotations="true"
      supportsImports="true"
      supportsQueryLanguage="ugcq">
  <SupportedRequest requestName="Retrieve"/>
  <SupportedRequest requestName="..."/>
  ...
</DIGDescription>
```

The core DIG 2.0 schema [10] already has the *DIGDescription* tag. But it does not contain attribute *supportsQueryLanguage*. Since we want to have this attribute for identifying fragment of query languages that supported by the DL reasoner, we have to add this attribute inside of the *DIGDescription*

tag. Due to the structure of the DIG 2.0 core schema, we can not extend the *DIGDescription* tag inside of the original DIG 2.0 core schema to have additional attribute.

To solve this problem, we can:

1. Define a new *DIGDescriptionExtension* tag that has exactly the same attributes as *DIGDescription* but with additional *supportsQueryLanguage* attribute.

2. Change the original core DIG 2.0 XML Schema as following below:

```
<xs:group name="Response">
   <xs:choice>
   ...
      <xs:element name="DIGDescription">
         <xs:complexType>
            <xs:sequence>
              <xs:element name="SupportedRequest"
              minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                   <xs:attribute name="requestName"
                   type="xs:anyURI"/>
                      </xs:complexType>
              </xs:element>
              </xs:sequence>
              <xs:attribute name="name" type="xs:string"/>
              <xs:attribute name="version" type="xs:string"/>
              <xs:attribute name="message" type="xs:string"/>
              <xs:attribute name="supportedLanguage" type="xs:string"/>
              <xs:attribute name="supportsAnnotations" type="xs:boolean"/>
              <xs:attribute name="supportsImports" type="xs:boolean"/>
              <xs:attribute name="supportsQueryLanguage" type="xs:string"/>
         </xs:complexType>
      </xs:element>
   </xs:choice>
</xs:group>
```

2.3.2 Retrieve Language

Retrieve language provides constructs to form requests to DL reasoners. The *Retrieve* request which is proposed in Query Answering Interface is derived from the core class *RequestToOntology* (see Figure 1.1). Multiple requests can be bundled into one *RequestMessage*. *Retrieve* has *ontologyURI*, *queryID* and *asID* as attributes. The *ontologyURI* is used to define the location of the ontology that is going to be used during the query answering process. The optional attribute *queryID* as unique query identification for query management. The *asID* is used by the client as an ID for iterative query answering that will be generated by the DL reasoner.

In the *Retrieve* statement, we can also use two more attributes, which are *ntuples* and *mode*. The attribute *ntuples* is used when the reasoner has to deal with large answer sets of tuples. The process which normally takes long time can be accelerated helps iterative query answering. For that, maximum number of tuples which are assumed to be returned can be specified.

In addition, DIG 2.0 Query Interface supports instructions to let a query answering engine compute results "proactively" to provide faster retrieval of subsequent chunks of tuples. This can be achieved by setting the optional *mode* attribute.

Retrieve statement consists of two parts, *QueryHead* and *QueryBody*. Within the *QueryHead* tag individuals and variables (denoted as *QueryVariable*) can be used. Variables are bound to those individuals which satisfy the query. When the query result is boolean, the *QueryHead* would be empty.

Complex queries are built from query atoms using boolean constructs for conjunction (*QueryObjectIntersectionOf*), union (*QueryObjectUnionOf*) and negation (*QueryObjectComplementOf*) (for the latter, for instance, negation as failure semantics is assumed). *ConceptQueryAtoms* consists of variables (or individuals) and complex concept expressions. *RoleQueryAtoms* consists of at least two identifiers for variables (or individuals) followed by a role expression.

The following conjunctive query asks for all individuals of the concept

`woman` which have `female` children. The requested knowledge base is identified by means of a URI.

```
<?xml version="1.0" encoding="UTF-8"?>
  <RequestMessage xmlns="http://dl.kr.org/dig2.0#"
  xmlns:owl11xml="http://www.w3.org/2006/12/owl11-xml#"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://dl.kr.org/dig2.0# dig2.0-ext.xsd
  http://www.w3.org/2006/12/owl11-xml# owl1.1.xsd ">
   <Retrieve queryID="q1" ontologyURI="www.ontologyURI" >
      <QueryHead>
         <QueryVariable URI="#x"/>
      </QueryHead>
      <QueryBody>
         <QueryObjectIntersectionOf>
            <ConceptQueryAtom>
              <QueryVariable URI="#x"/>
                 <owl11xml:ObjectIntersectionOf>
                    <owl11xml:OWLClass owl11xml:URI="#woman"/>
                    <owl11xml:ObjectSomeValuesFrom>
                       <owl11xml:ObjectProperty owl11xml:URI="#hasChild"/>
                       <owl11xml:OWLClass owl11xml:URI="#female"/>
                    </owl11xml:ObjectSomeValuesFrom>
                 </owl11xml:ObjectIntersectionOf>
            </ConceptQueryAtom>
         </QueryObjectIntersectionOf>
      </QueryBody>
   </Retrieve>
  </RequestMessage>
```

We use DIG namespace as default namespace in our schema. If in the future the Query Answering Interface will be part of DIG 2.0 core schema, then we do not have to use other namespaces for our extension.

Based on the assumption and for simplification, we eliminate all namespace and its prefix from all examples presented in this report. However, since Query Answering Interface is actually still an extension of DIG 2.0, we have

to add the appropriate namespace. We use *qai* (as abbreviation from Query Answering Interface) as prefix.

```
<?xml version="1.0" encoding="UTF-8"?>
  <RequestMessage
  xmlns="http://dl.kr.org/dig2.0#"
  xmlns:qai="http://extension/QueryAnswers"
  xmlns:owl11xml="http://www.w3.org/2006/12/owl11-xml#"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://dl.kr.org/dig2.0# dig2.0-ext.xsd
  http://www.w3.org/2006/12/owl11-xml# owl1.1.xsd
  http://extension/QueryAnswers ExtensionQueryAnswers.xsd">
   <Retrieve qai:queryID="q1" qai:ontologyURI="www.ontologyURI" >
      <qai:QueryHead>
         <qai:QueryVariable qai:URI="#x"/>
      </qai:QueryHead>
      <qai:QueryBody>
         <qai:QueryObjectIntersectionOf>
            <qai:ConceptQueryAtom>
              <qai:QueryVariable qai:URI="#x"/>
                 <owl11xml:ObjectIntersectionOf>
                    <owl11xml:OWLClass owl11xml:URI="#woman"/>
                    <owl11xml:ObjectSomeValuesFrom>
                       <owl11xml:ObjectProperty owl11xml:URI="#hasChild"/>
                       <owl11xml:OWLClass owl11xml:URI="#female"/>
                    </owl11xml:ObjectSomeValuesFrom>
                 </owl11xml:ObjectIntersectionOf>
            </qai:ConceptQueryAtom>
         </qai:QueryObjectIntersectionOf>
      </qai:QueryBody>
   </qai:Retrieve>
  </qai:RequestMessage>
```

In the XML document above, the client send a *Retrieve* request to the DL reasoner. The *Retrieve* request is encapsulated inside the *RequestMessage* element.

Since we are using some of the elements that located in others XML

schemas, we have to include their namespaces too. The declaration of namespaces is located in the root element (*RequestMessage*).

These namespaces refer to the appropriate datatypes. Each of the namespace has its own prefix (e.g. owl11xml, qai, xsi). This prefix makes the syntax in the XML document more readable.
E.g., we can just write `owl11xml:URI=“#female”` in a short way.

2.3.3 Tell Language

In order to tell the reasoner that no more tuples will be requested, the statement *ReleaseQuery*, which has an attribute *queryID* is send. The reasoner should respond with the *Confirmation* response.

```
<ReleaseQuery queryID="q1"/>
```

2.3.4 Response Language

The *QueryAnswers* response to a *Retrieve* request has an attribute *queryID* which corresponds with the id of the submitted query. The answer set id *asID* will be generated by the reasoner. The response contains tuples of bindings for variables mentioned in the *QueryHead*. The response head is the same as the head of the corresponding query. The head is inserted just for convenience; the reasoner may not reorder components of bindings. For example, below we can see the response to the query posed above (see Section 2.3.2):

```
<QueryAnswers queryID="q1" asID="asid1000">
  <QueryHead>
     <QueryVariable URI="#x"/>
  </QueryHead>
  <Binding>
     <owl11xml:Individual owl11xml:URI="#mary"/>
  </Binding>
  <Binding>
     <owl11xml:Individual owl11xml:URI="#susan"/>
```

```
    </Binding>
  </QueryAnswers>
```

2.4 XML Schema for Query Answering Interface

In this section, we discuss the design and application scenarios of the proposed XML Schema for the Query Answering Interface in DIG 2.0.

2.4.1 Design

As the DIG interface is considered as a standard interface for accessing DL reasoning systems, it is a desirable goal not only to standardize the core DIG but also its extensions. However, it is not possible to put all kinds of useful extensions and operators into the DIG specification in advance because of evolving and upcoming work in the context of non-standard inference services and reasoning related services.

Therefore to enable the embedding of new requests and responses into XML messages, implementors should use the redefinition mechanism of XML Schema and extend the *Request* and *Response* element groups from the core DIG 2.0 schema.

The XML schema we designed for Query Answering Interface is distributed over following files:

- DIG2.0-ext.xsd (aka RedefineDIGSchema): This is the file where we extend the core schema of DIG 2.0 (see Appendix A.1).

- ExtensionQueryAnswers.xsd (aka ExtensionSchema): This is the core schema of Query Answering Interface.

Besides those files, we also used some of standard XML Schemas below:

- DIG2.0.xsd (aka DIGSchema): This is the core schema of DIG 2.0.

- OWL1.1.xsd (aka OWLSchema): This is the core schema of OWL 1.1.
- XML.xsd (aka XMLSchema): This is the schema of XML namespace.

First of all we need to extend the core of DIG 2.0 Schema. This redefinition is located in the RedefineDIGSchema file. We redefine the group named *Request* from DIGSchema and extend them with two new elements: *Retrieve* and *ReleaseQuery*. In this way, we can use besides other elements also *Retrieve* and *ReleaseQuery* tags inside of the *RequestMessage* tag.

This file RedefineDIGSchema has the same namespace as the DIGSchema. This is due to the redefinition mechanism of XML Schema: the namespace of the redefining schema must be the same as the namespace of the redefined schema.

To redefine the DIGSchema, we use a *redefine* element. The *redefine* element allows for redefinition of simple and complex types, groups, and attribute groups from an external schema.

```
<xs:redefine schemaLocation="dig2.0.xsd">
   <xs:group name="Request">
      <xs:choice>
         <xs:group ref="Request"/>
         <xs:element name="Retrieve" type="RetrieveType"/>
         <xs:element name="ReleaseQuery" type="ReleaseType"/>
      </xs:choice>
   </xs:group>
</xs:redefine>
```

To make use of some of tags (e.g., OWLClass) that already exist in other schemas, we need the *import* tag, which allows us to add multiple schemas with different target namespace to a document. In our work, we imported OWLSchema, XMLSchema and DIGSchema.

```
<xs:import namespace="http://www.w3.org/2006/12/owl11-xml#"
   schemaLocation="owl1.1.xsd"/>
<xs:import namespace="http://www.w3.org/XML/1998/namespace"
   schemaLocation="http://www.w3.org/2001/xml.xsd"/>
```

```
<xs:import namespace="http://dl.kr.org/dig2.0#"
   schemaLocation="dig2.0.xsd"/>
```

The ExtensionSchema consists of several parts. We defined several complex types, which are e.g., *RetrieveType* for specifying the *Retrieve* element, *ReleaseType* for *Release* element, *QueryAnswerType* for *QueryAnswers* element, etc.

The code below defines the complex type *RetrieveType* which has four attributes: *queryID*, *ntuples*, *asID* and *mode*.

```
<xs:complexType name="RetrieveType">
   <xs:complexContent>
      <xs:extension base="RequestToOntology"/>
         <xs:attribute name="queryID" type="xs:string" use="required"/>
         <xs:attribute name="ntuples" type="xs:string" use="optional"/>
         <xs:attribute name="asID" type="xs:string" use="optional"/>
         <xs:attribute name="mode" type="xs:string" use="optional"/>
         <xs:sequence>
            <xs:group ref="RetrieveGroup" minOccurs="1"/>
         </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

After the XML Schema is defined, we can proceed with the next step, the validation of XML documents against the schema. For this, we tried different validation methods. To do the validation programmatically, we used XMLBeans. We can use XMLBeans not only to validate the XML documents against the XML Schema, but also to modify the XML data by means of programms.

2.4.2 XMLBeans

XMLBeans [21] developed by Apache Software allows us to access the full power of XML in a Java friendly way. The behind idea of this technology is that we can get the features of XML and XML Schema and have these

features mapped as naturally as possible to the equivalent Java language and typing constructs.

XMLBeans uses XML Schema to compile Java interfaces and classes that can be used to access and modify XML instance data. Using XMLBeans is similar to using any other Java interface/class. While a major use of XMLBeans is to access our XML instance data with strongly typed Java classes there are also API's that allow access to the full XML infoset (XMLBeans keeps XML infoset fidelity) as well as to allow reflecting into the XML Schema itself through an XML Schema Object model.

XMLBeans was chosen because it supports full XML Schema, which is critical if we want to have control over the features of the XML Schema when working with Java environment. After installing XMLBeans we can just use command "validate" that has two input parameters: the XML document to be validated and the XML Schema.

```
validate XMLSchema.xsd FileToValidate.xml
```

The result of validation procedure is printed to the standard output.

Besides XMLBeans, there are also others online and offline tools for validating the XML Schema. One of them we have used is the online validator of DecisionSoft [19].

With XMLBeans, not only we can do validation, but we can also manipulate objects of the XML Schema Object model through the Java classes. We can generate Java types using XMLBeans that represent schema types. In this way, we can access instances of the schema through JavaBeans-style accessors. The XMLBeans API also allows us to reflect into the XML Schema itself through an XML Schema Object model. To make use of this object model, we have to generate the jar file from our XML Schema. This can be done by applying the compilation command "scomp" to the schema.

```
scomp -out file.jar XMLSchema.xsd
```

After that, we can extend the generated Java code if required. In order to do this, we have to import several packages first (including the jar of our schema denoted below as `extension.*`):

```
// The jar file of all our extended schema including the imported file
import extension.*;
import org.apache.xmlbeans.XmlException;
import org.apache.xmlbeans.impl.regex.REUtil;
```

After that we can use classes and methods generated by the XMLBeans for parsing a XML document and manipulating data provided by the parsed XML document. In the following example, we created the class *QueryAnswering* for parsing the XML document located in c:/workspace/Redefine.xml, adding a new query variable to the query head and printing out the modified XML document.

```
public class QueryAnswering {
   public static void main(String[] args) {
      File xmlFile = new File("c:\\workspace\\Redefine.xml");
//  Bind the instance to the generated XMLBeans types.
      try {
      RequestMessageDocument requestMessageDocument =
      RequestMessageDocument.Factory.parse(xmlFile);
      RequestMessageDocument.RequestMessage requestMessage =
      requestMessageDocument.getRequestMessage();
      RetrieveType retrieve = requestMessage.getRetrieveArray(0);
      QueryHeadType queryHead = retrieve.getQueryHead();
      Variable variable = queryHead.addNewQueryVariable();
      variable.setURI("#y"); //adding new variable
      System.out.println(requestMessageDocument);
...}
```

The proposed XML Schema for Query Answering Interface has been tested using the online XML Schema validator from DecisionSoft, and also using XMLBeans. This schema has been also tested by one of the Rika Project groups from STS department of Hamburg University of Technology.

Chapter 3

Translation of OWL-QL to/from DIG 2.0

3.1 Motivation

At the moment, there is any server that supports DIG 2.0 protocol. The development is still work on progress. But there are already existing servers that support other standard query languages. One of such query languages is OWL-QL [14], a candidate standard language for DL. To make use the existing OWL-QL server compatible with the DIG 2.0 Query Answering Interface, we propose translation rules to transform OWL-QL queries into DIG 2.0 queries using some translation module that have to be implemented. The translation is possible, since both query languages have an overlap considering their semantics.

3.2 XSLT

XSLT (eXtensible Stylesheet Language Transformation) is a mechanism that is used for transforming document in XML format into other useful formats (e.g., into HTML or XHTML) or between different XML schemas [13]. XSLT does not change the original document, the one that used as input document;

rather, a new document as output is created based on the existing one. To do the translation, an XSLT processor is needed. There are several processors that can be used. Xalan [20] and Saxon [15] are two of the widely used XSLT processors at the moment. The XSLT processor takes two input files - an XML source document, and an XSLT stylesheet - and produces an output document. The XSLT stylesheet contains collection of template rules that the processor applies to the input document during the transformation procedure.

Two main aspects in the transformation process are:

- The first stage is a structural transformation, in which the data is converted from the structure of the incoming XML document to a structure that reflects the desired output.
- The second stage is formatting, in which the new structure is outputed in the required format such as HTML or PDF.

XSLT makes use of XPath [5]. XPath is a non-XML language used to identify particular parts of XML document. XPath can be described in its three crucial roles. First, it is used within the XSLT stylesheet for addressing parts of the input document. Second, XPath is used as a pattern language in the matching rules inside of the stylesheet. Third, it is used to perform simple math and string manipulations via built-in XPath operators and functions.

In the next sections, we first give a short overview of OWL-QL and then present translation rules.

3.3 OWL-QL

OWL-QL [14] is a formal language and protocol for a querying knowledge bases represented in OWL. OWL-QL specifies the semantic relationships among a query, a query answer, and the KBs used to produce the answer. OWL-QL supports query-answering dialogues in which the answering agent may use automated reasoning methods to derive answers to queries, as well

as scenarios in which the knowledge to be used in answering a query may be in multiple knowledge bases on the Semantic Web, and/or where those knowledge bases are not specified by the client.

An OWL-QL query-answering dialogue is initiated by a client sending a query to an OWL-QL server. An OWL-QL query is an object necessarily containing a query pattern that specifies a collection of OWL sentences in which some URIrefs are considered to be variables.

A query may have zero or more answers, each of which provides bindings of URIrefs or literals to some of the variables in the query pattern. Each request from a client to a server for answers to a query can include an answer bundle size bound, and the server is required to respond by delivering an answer bundle containing at most the number of query answers given by the answer bundle size bound.

The set of OWL sentences that are used by the server in answering a query is referred to as the answer KB. An OWL-QL query contains an answer KB pattern that is a KB, a list of KB references, or a variable.

In what follows, we give an example of OWL-QL query, that requests to "find 4 individuals (bindings of the variable x) that are of type GraduateStudent and takes GraduateCourse0 as course".

```
<?xml version="1.0" encoding="UTF-8"?>
<owl-ql:query xmlns:owl-ql="http://www.w3.org/2003/10/owl-ql-syntax#"
   xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:uni="http://www.uni.edu/univ-bench.owl#">
   <owl-ql:queryPattern>
    <rdf:RDF>
     <rdf:Description
      rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
      <rdf:type
      rdf:resource="http://www.uni.edu/univ-bench.owl#GraduateStudent"/>
      </rdf:Description>
     <rdf:Description
      rdf:about="http://www.w3.org/2003/10/owl-ql-variables#x">
```

```
      <uni:takesCourse
      rdf:resource="http://www.Department0.edu/GraduateCourse0"/>
     </rdf:Description>
    </rdf:RDF>
   </owl-ql:queryPattern>
   <owl-ql:mustBindVars>
     <var:x/>
  </owl-ql:mustBindVars>
  <owl-ql:answerKBPattern>
    <owl-ql:kbRef rdf:resource=
     "file://localhost/c:/Programme/RacerPro/university0.owl"/>
  </owl-ql:answerKBPattern>
  <owl-ql:answerSizeBound> 4 </owl-ql:answerSizeBound>
</owl-ql:query>
```

3.4 Translation Rules

To do the translation between OWL-QL and DIG 2.0, XSLT translation rules are required. The translation procedure can be performed by some translation module that will be integrated into an OWL-QL server (e.g., RacerManager) or can be used as stand alone application.

In this context, we refer to RacerManager [4] as one of OWL-QL sever. RacerManager is an open-source Semantic Web middleware that serves as a scalable front-end for applications to efficiently query OWL ontologies. RacerManager has been developed to serve as an OWL-QL application server for the DL reasoner RacerPro. The RacerManager delegates queries to RacerPro servers that manage the KBs mentioned in the query and load KBs on demand. The system focuses on techniques which allow to achieve better scalability, high availability and the required quality of service by implementing such techniques as query dispatching, load balancing and caching of query answers. To improve usability of RacerManager we can equip it with the translation module mentioned above which would transform OWL-QL queries sent to the DL reasoner into DIG 2.0 queries and answers sent from

DL reasoner into OWL-QL variable bindings.

The XSLT stylesheet for OWL-QL to/from DIG 2.0 schema translation is divided into two parts: request rules and response rules. We start with request rules, which are used to translate OWL-QL query into DIG 2.0 query. According to the OWL-QL syntax, the query body has to be formed inside of the *queryPattern* tag. In the query head, the tag *mustBindVars* is used to specify variables that must be bound to available answers. The URL of the KB to be used for answering the query is specified in the *answerKBPattern* tag. The *answerSizeBound* defines the maximum number of results the client wants to get for this query.

We refer to example in the Section 3.3 to show some of defined translation rules defined in this work. Below is the snippet of the stylesheet file defining the translation rule for the element *mustBindVars*. Here we postulate that every variable mentioned inside of *mustBindVars* tag has to be transformed into the variable with the same name inside of the *QueryHead* tag in DIG 2.0. The *queryPattern* tag in OWL-QL will be translated to the *QueryBody* tag in DIG 2.0. We translate subelements of *queryPattern* tag (e.g., *Description*) to subelements of the *QueryBody* tag in DIG 2.0 (e.g., *ConceptQueryAtom*), etc.

```
   <xsl:stylesheet version="2.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns="http://dl.kr.org/dig2.0#"
     xmlns:owl11xml="http://www.w3.org/2006/12/owl11-xml#"
     xmlns:qai="http://extension/QueryAnswers"
     exclude-result-prefixes="var owl-ql rdf ">
//We define the namespaces that we are going to use
     <xsl:output method="xml" indent="yes" encoding="utf-8"/>
//The output method type
     <xsl:template match="owl-ql:mustBindVars">
        <QueryHead>
           <xsl:element name="QueryVariable">
              <xsl:attribute name="URI">
                 <xsl:value-of select="concat('#',local-name(*))"/>
              </xsl:attribute>
```

```
            </xsl:element>
         </QueryHead>
      </xsl:template>
<xsl:template match="owl-ql:queryPattern">
      <QueryBody>
         <xsl:for-each select="//rdf:Description/@rdf:about">
            <ConceptQueryAtom>
               <QueryVariable>
                  <xsl:attribute name="URI" >
                     <xsl:copy-of select="."/>
                  </xsl:attribute>
               </QueryVariable>
               <owl11xml:OWLClass>
                  <xsl:attribute name="owl11xml:URI">
                     <xsl:value-of select="//rdf:type/@rdf:resource"/>
                  </xsl:attribute>
               </owl11xml:OWLClass>
            </ConceptQueryAtom>
         </xsl:for-each>
         ...
      </QueryBody>
  </xsl:template>
```

As mentioned before, the XSLT processor translates queries from OWL-QL to DIG 2.0 using translation rules, e.g., one defined above. The result of the translation can be seen below. The translated query is now encapsulated in *RequestMessage* tag. It has *QueryHead* consisting of zero or more *QueryVariable* tags. The query body defined before inside of *mustBindVars* (OWL-QL) now translated into *QueryBody* (DIG 2.0).

```
<?xml version="1.0" encoding="utf-8"?>
   <RequestMessage xmlns="http://dl.kr.org/dig2.0#"
     xmlns:owl11xml="http://www.w3.org/2006/12/owl11-xml#"
    xmlns="http://extension/QueryAnswers"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <Retrieve queryID="223"
       ontologyURI="file://localhost/c:/Programme/RacerPro/university0.owl"
```

```
    ntuples="4">
     <QueryHead>
       <QueryVariable URI="#x"/>
     </QueryHead>
     <QueryBody>
       <ConceptQueryAtom>
          <QueryVariable
           URI="http://www.w3.org/2003/10/owl-ql-variables#x"/>
          <owl11xml:OWLClass
           owl11xml:URI="http://www.uni.edu/univ-bench.owl#GraduateStudent"/>
       </ConceptQueryAtom>
       ...
     </QueryBody>
   </Retrieve>
</RequestMessage>
```

After the DL Reasoner received the query, it will send its response to the client in DIG 2.0.

```
<ResponseMessage xmlns="http://dl.kr.org/dig2.0#"
 xmlns:="http://extension/QueryAnswers"
 xmlns:owl11xml="http://www.w3.org/2006/12/owl11-xml#"
 xmlns:uni="http://www.uni.edu/univ-bench.owl#">
  <QueryAnswers queryID="223" asID="abc123">
    <QueryHead>
      <QueryVariable URI="#x"/>
    </QueryHead>
    <Binding >
      <owl11xml:Individual
       owl11xml:URI="http://www.Department0.edu/GraduateStudent44"/>
    </Binding>
  </QueryAnswers>
  ...
</ResponseMessage>
```

To translate the response from DIG 2.0 into the OWL-QL, we proposed response translation rules. In the following example for response rules we declare that every *QueryAnswers* element found in the input file will be

translated into the *answerBundle* element of OWL-QL. For that, we also use templates *Binding* and *notifier* presented below. The *Binding* tag itself will be translated into the *answer* tag, and every answer tuple will be translated into the corresponding *binding-set* of elements. We obtain names of elements from the *QueryHead*.

```
<xsl:template match="QueryAnswers">
   <owl-ql:answerBundle>
      <xsl:apply-templates select="Binding"/>
      <xsl:apply-templates select="notifier"/>
   </owl-ql:answerBundle>
</xsl:template>
<xsl:variable name="var1"
 select="substring-after
 (//qai:QueryHead/qai:QueryVariable/qai:@URI,'#')"/>
<xsl:variable name="var2" select="concat('var:',$var1)"/>
<xsl:template match="Binding">
   <owl-ql:answer>
     <owl-ql:binding-set>
        <xsl:element name="{$var2}">
           <xsl:attribute name="rdf:resource" >
              <xsl:value-of
               select="//owl11xml:Individual/@owl11xml:URI"/>
           </xsl:attribute>
        </xsl:element>
     </owl-ql:binding-set>
</xsl:template>
<xsl:template match="notifier">
   <owl-ql:continuation>
      <xsl:choose>
         <xsl:when test="@message='last'">
           <owl-ql:termination-token>
             <xsl:choose>
                 <owl-ql:end/>
             </xsl:choose>
           </owl-ql:termination-token>
         </xsl:when>
         <xsl:otherwise>
```

```
            <owl-ql:continuation-token>
              <owl-ql:processHandle>
                 <xsl:value-of select="//QueryAnswers/@asID"/>
              </owl-ql:processHandle>
               <owl-ql:answerBundleSize>
                 <xsl:value-of select="//notifier/@message"/>
              </owl-ql:answerBundleSize>
            </owl-ql:continuation-token>
          </xsl:otherwise>
       </xsl:choose>
    </owl-ql:continuation>
</xsl:template>
```

After performing translation we receive e.g. the following output:

```
<owl-ql:answer>
  <owl-ql:binding-set>
   <var:x rdf:resource="http://www.Department0.edu/GraduateStudent44"/>
  </owl-ql:binding-set>
  <owl-ql:answerPatternInstance>
    <rdf:RDF xmlns:uni="http://www.uni.edu/univ-bench.owl#"
    xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#">
     <rdf:Description
      rdf:about="http://www.Department0.edu/GraduateStudent44">
      <rdf:type
      rdf:resource="http://www.uni.edu/univ-bench.owl#GraduateStudent"/>
      <uni:takesCourse
      rdf:resource="http://www.Department0.edu/GraduateCourse0"/>
     </rdf:Description>
    </rdf:RDF>
  </owl-ql:answerPatternInstance>
</owl-ql:answer>
```

We have tested presented rules using XSLT processors from Saxon and Xalan.

3.5 Limitation Of The Translation Rules

The problem that we are facing during defining translation rules is that not all elements are described in both XML schemas (OWL-QL and DIG 2.0) (see 3.4). To overcome this limitation, we made several assumptions:

- The *queryID* attribute used in the *Retrieve* tag of DIG 2.0 is not available in OWL-QL. We proposed to obtain the value of the *queryID* to uniquely generated by the client sent the OWL-QL query.
- The *asID* attribute (Answer Set ID) is proposed to be used as substitution of the *processHandler* in OWL-QL response. This attribute is essential for the iterative query answering. *asID* is generated by DL reasoners and can be used by OWL-QL clients for subsequent queries. Using *asID*, the DL reasoner will send the rest of the response according to the client request (e.g., the client can ask for n more tuples or the rest of the response).
- The *continuation* tag in OWL-QL has two possible values:
  - *termination-token* tag can be: *none* or *end*. The tag *none* means that the server already sent all the answers that known. And the *end* tag means that the server had gave all the answers. Since DIG does not have the appropriate tag to perform the substitution, we propose to use the *message* attribute of the *notifier* tag. The value of the message can be restricted to some meaningful definitions. For example, when the value is *last*, than it will be considered as *end* or *none* tag in OWL-QL.
  - inside of the *continuation-token* tag the *answerBundleSize* has to be specified. Since DIG 2.0 does not have an equal tag for adequate translation, we assume that the value of *answerBundleSize* will be sent by DL reasoner within the *notifier message*.

  To support this, the messaging/notifier mechanism of DIG 2.0 has to be extended. We have to declare some constraints for the value of

notifier message. At the moment, *last* can be used for declaring `end` and `none` value of the `termination-token`. Omitting this message or having other values for message means `continuation-token`.

Chapter 4

Conclusion

4.1 Summary

DIG 2.0 extension for Query Answering is currently of much interest in the DL community. To apply this extension, we proposed the XML Schema that can be used for validating and parsing XML documents containing queries and answers to the queries. To use the Query Answering XML schema, we need to declare the namespace of the schema that we are referring to and its physical location.

The schema is needed as a frame to be used to validate the XML document. But the schema itself is not enough. There are constraints which cannot or are to complicated to be expressed (e.g., for defining complex value of the *notifier message*) with XML Schemas. We can overcome those problems with writing some additional Java, Perl, C++, etc code to check additional constraints. We can use the Java classes that generated by the XMLBeans and add some constraints to the schema.

We also made translation rules, that can help the existing OWL-QL implementations to support communication with DIG 2.0 reasoners. When defining translation rules, we had to make some assumptions to overcome the problem, that a number of constructs in OWL-QL lacks of suitable substitution in DIG 2.0. Since we only do the translation of query answers from

DIG to OWL-QL, the only workaround we had to propose was considering the query id and answer set id.

4.2 Further Work

The Query Answering XML schema that we proposed here is still an extension of the core DIG 2.0 schema. This results in a number of schemas that we have to use (e.g., the core DIG 2.0 schema, the extension schema for Query Answering, and the redefinition schema). When our proposal is already integrated into the core DIG 2.0 schema, it will be possible to reduce the number of namespace that have to be referred in DIG documents.

We already have defined translation rules that can be used in the future to build some translation module. With this module, an OWL-QL server will translate OWL-QL to DIG 2.0 and vice versa. The translation can be very useful if there is already DL reasoner that supports DIG 2.0.

Appendix A

Query Answering Schema

A.1 Redefinition (Extension) of Core DIG Schema

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <xs:schema targetNamespace="http://dl.kr.org/dig2.0#"
   xmlns="http://dl.kr.org/dig2.0#"
   xmlns:qai="http://extension/QueryAnswers"
   xmlns:owl11xml="http://www.w3.org/2006/12/owl11-xml#"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   elementFormDefault="qualified"
   attributeFormDefault="qualified">

   <xs:import namespace="http://extension/QueryAnswers"
    schemaLocation="ExtensionQueryAnswers.xsd"/>
   <xs:import namespace="http://www.w3.org/2006/12/owl11-xml#"
    schemaLocation="owl1.1.xsd"/>

   <xs:redefine schemaLocation="dig2.0.xsd">
   <!-- Redefine REQUEST -->
     <xs:group name="Request">
        <xs:choice>
           <xs:group ref="Request"/>
           <xs:element name="Retrieve" type="qai:RetrieveType"/>
           <xs:element name="ReleaseQuery" type="qai:ReleaseType" />
        </xs:choice>
```

```
    </xs:group>

    <!-- Redefine Response -->
    <xs:group name="Response">
       <xs:choice>
          <xs:group ref="Response"/>
          <xs:element name="QueryAnswers"
          type="qai:QueryAnswersType"/>
          <xs:element name="DIGDescriptionExtension"
          type=qait:DIGDescriptionExtensionType"/>
       </xs:choice>
    </xs:group>
  </xs:redefine>
</xs:schema>
```

A.2 XML Schema of Query Answering

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <xs:schema targetNamespace="http://extension/QueryAnswers"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns="http://extension/QueryAnswers"
   xmlns:owl11xml="http://www.w3.org/2006/12/owl11-xml#"
   xmlns:dig="http://dl.kr.org/dig2.0#"
   elementFormDefault="qualified"
   attributeFormDefault="qualified">

   <xs:import namespace="http://www.w3.org/2006/12/owl11-xml#"
    schemaLocation="owl1.1.xsd"/>
   <xs:import namespace="http://www.w3.org/XML/1998/namespace"
    schemaLocation="http://www.w3.org/2001/xml.xsd"/>
   <xs:import namespace="http://dl.kr.org/dig2.0#"
    schemaLocation="dig2.0.xsd"/>

   <xs:complexType name="RetrieveType">
      <xs:complexContent>
        <xs:extension base="dig:RequestToOntology">
          <xs:attribute name="queryID" type="xs:string"
```

```
        use="required"/>
        <xs:attribute name="ntuples" type="xs:string"
        use="optional"/>
        <xs:attribute name="mode" type="xs:string"
        use="optional"/>
        <xs:attribute name="asID" type="xs:string"
        use="optional"/>
        <xs:sequence>
           <xs:group ref="RetrieveGroup" minOccurs="1"/>
        </xs:sequence>
      </xs:extension>
   </xs:complexContent>
</xs:complexType>

<xs:group name="RetrieveGroup">
   <xs:sequence>
      <xs:element name="QueryHead" type="QueryHeadType"
      minOccurs="0"/>
      <xs:element name="QueryBody" type="QueryBodyType"
      minOccurs="0"/>
   </xs:sequence>
</xs:group>

<xs:complexType name ="QueryHeadType">
   <xs:sequence>
      <xs:element ref ="QueryVariable" minOccurs="0"
      maxOccurs="unbounded"/>
      <xs:element ref ="ConcreteDomainQueryVariable"
      minOccurs="0" maxOccurs="unbounded"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name ="QueryBodyType">
   <xs:sequence>
      <xs:group ref="BooleanConstructGroup" minOccurs="0"
       maxOccurs="unbounded"/>
      <xs:group ref="QueryAtomsGroup" maxOccurs="unbounded"/>
   </xs:sequence>
```

```
</xs:complexType>

<xs:group name="BooleanConstructGroup">
   <xs:choice>
      <xs:element name="QueryObjectIntersectionOf"/>
      <xs:element name="QueryObjectUnionOf"/>
      <xs:element name="QueryObjectComplementOf" />
   </xs:choice>
</xs:group>

<xs:group name="QueryAtomsGroup">
   <xs:choice>
      <xs:element ref="ConceptQueryAtom" minOccurs="0"/>
      <xs:element ref="RoleQueryAtom" minOccurs="0"/>
      <xs:element ref="ConcreteDomainQueryAtom" minOccurs="0"
       maxOccurs="unbounded"/>
      <xs:element ref="PredicateQueryAtom" minOccurs="0"/>
      <xs:element ref="SameAsQueryAtom" minOccurs="0"/>
      <xs:element ref="DifferentFromQueryAtom" minOccurs="0"/>
      <xs:element ref="QueryProject" minOccurs="0"/>
   </xs:choice>
</xs:group>

<xs:element name="ConceptQueryAtom">
   <xs:complexType>
      <xs:sequence>
         <xs:group ref="VarIndv" />
         <xs:group ref="owl11xml:Description" />
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="RoleQueryAtom">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="owl11xml:ObjectProperty"/>
         <xs:group ref="VarIndv" minOccurs="2"
         maxOccurs="unbounded"/>
```

```
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="ConcreteDomainQueryAtom">
   <xs:complexType>
      <xs:sequence>
         <xs:group ref="VarIndv"/>
         <xs:element ref ="ConcreteDomainQueryVariable"/>
         <xs:element ref="owl11xml:DataProperty"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="PredicateQueryAtom">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref ="ConcreteDomainQueryVariable"/>
         <xs:element ref ="owl11xml:Constant"/>
         <xs:element ref="predicate" minOccurs="0"
          maxOccurs="unbounded"/>
         <xs:element ref="lambda" minOccurs="0"
          maxOccurs="unbounded"/>
         <xs:element ref="op" minOccurs="0"
          maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:element name="QueryProject">
   <xs:complexType>
      <xs:choice>
         <xs:group ref="RetrieveGroup"/>
      </xs:choice>
   </xs:complexType>
</xs:element>

<xs:element name="SameAsQueryAtom">
```

```
   <xs:complexType>
      <xs:choice>
         <xs:element ref="QueryVariable" minOccurs="2"
          maxOccurs="unbounded"/>
         <xs:element ref="owl11xml:Individual" minOccurs="2"
          maxOccurs="unbounded"/>
      </xs:choice>
   </xs:complexType>
</xs:element>

<xs:element name="DifferentFromQueryAtom">
   <xs:complexType>
      <xs:choice>
         <xs:element ref="QueryVariable" minOccurs="2"
          maxOccurs="unbounded"/>
         <xs:element ref="owl11xml:Individual" minOccurs="2"
          maxOccurs="unbounded"/>
      </xs:choice>
   </xs:complexType>
</xs:element>

<xs:element name="op">
   <xs:complexType>
      <xs:simpleContent>
         <xs:extension base ="xs:string">
            <xs:attribute name="name" use="required"
            type="opAttributes"/>
         </xs:extension>
      </xs:simpleContent>
   </xs:complexType>
</xs:element>

<xs:simpleType name = "opAttributes">
   <xs:restriction base = "xs:string">
      <xs:enumeration value="&gt;"/>
      <xs:enumeration value="&lt;"/>
      <xs:enumeration value="="/>
   </xs:restriction>
```

```
    </xs:simpleType>

    <xs:complexType name="Variable">
       <xs:attribute name="URI" type="xs:anyURI" use="required"/>
       <xs:attributeGroup ref="xml:specialAttrs"/>
    </xs:complexType>

    <xs:group name="VarIndv">
       <xs:choice>
          <xs:element ref="QueryVariable"  minOccurs="0"/>
          <xs:element ref="owl11xml:Individual" minOccurs="0"/>
       </xs:choice>
    </xs:group>

    <xs:complexType name="BooleanConstructType">
       <xs:choice>
          <xs:group ref="BooleanConstructGroup" minOccurs="0"/>
       </xs:choice>
    </xs:complexType>

    <xs:element name="predicate" type="Variable"/>
    <xs:element name="lambda" type="Variable"/>
    <xs:element name="QueryVariable" type="Variable"/>
    <xs:element name="ConcreteDomainQueryVariable" type="Variable"/>

<!-- QueryAnswers start -->
    <xs:complexType name="QueryAnswersType">
       <xs:sequence>
          <xs:element name="QueryHead" type="QueryHeadType" minOccurs="0"/>
          <xs:element ref="Binding" minOccurs="0"/>
          <xs:element name="notifier" type="notifierType" minOccurs="0"/>
       </xs:sequence>
       <xs:attribute name="queryID" type="xs:string" use="required"/>
       <xs:attribute name="asID" type="xs:string" use="optional"/>
    </xs:complexType>

    <xs:element name="Binding">
       <xs:complexType>
```

```
            <xs:choice>
               <xs:group ref="QueryVariableGroup"/>
               <xs:group ref="ConcreteDomainQueryVariableGroup"/>
            </xs:choice>
         </xs:complexType>
      </xs:element>

      <xs:group name="QueryVariableGroup">
         <xs:sequence>
            <xs:element name="QueryVariable" type="Variable"
             minOccurs="0"/>
            <xs:element ref="owl11xml:Individual" minOccurs="0"/>
         </xs:sequence>
      </xs:group>

      <xs:group name="ConcreteDomainQueryVariableGroup">
         <xs:sequence>
            <xs:element ref="ConcreteDomainQueryVariable"
             minOccurs="0"/>
            <xs:element ref="ConcreteDomainBinding"
             minOccurs="0"/>
         </xs:sequence>
      </xs:group>

      <xs:element name="ConcreteDomainBinding">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="cdvar" type="Variable"/>
               <xs:element ref="owl11xml:Constant" minOccurs="0"/>
            </xs:sequence>
         </xs:complexType>
      </xs:element>

      <xs:complexType name="notifierType">
         <xs:attribute name="message" type="xs:string"/>
      </xs:complexType>

<!-- Release Begin -->
```

```
  <xs:complexType name="ReleaseType" >
     <xs:attribute name="queryID" type="xs:int" use="optional"/>
  </xs:complexType>

  <xs:complexType name="DIGDescriptionExtensionType">
     <xs:sequence>
        <xs:element name="SupportedRequest" minOccurs="0"
         maxOccurs="unbounded">
           <xs:complexType>
              <xs:attribute name="requestName" type="xs:anyURI"/>
           </xs:complexType>
        </xs:element>
     </xs:sequence>
     <xs:attribute name="dig:name" type="xs:string"/>
     <xs:attribute name="dig:version" type="xs:string"/>
     <xs:attribute name="dig:message" type="xs:string"/>
     <xs:attribute name="supportedLanguage" type="xs:string"/>
     <xs:attribute name="supportsAnnotations" type="xs:boolean"/>
     <xs:attribute name="supportsImports" type="xs:boolean"/>
     <xs:attribute name="supportsQueryLanguage" type="xs:string"/>
  </xs:complexType>
</xs:schema>
```

Appendix B

XSLT Rules for OWL-QL and DIG 2.0

B.1 Request

```
  <?xml version="1.0" encoding="utf-8"?>
  <xsl:stylesheet version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
   xmlns:owl-ql="http://www.w3.org/2003/10/owl-ql-syntax#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns="http://dl.kr.org/dig2.0#"
   xmlns:owl11xml="http://www.w3.org/2006/12/owl11-xml#"
   xmlns:qai="http://extension/QueryAnswers"
   <exclude-result-prefixes="var owl11xml-ql rdf ">

   <xsl:output method="xml" indent="yes" encoding="utf-8"/>

  <xsl:template match="owl-ql:query" >

   <RequestMessage>
      <xsl:variable name="x" select="." />
<!-- create namespace declarations -->
         <xsl:for-each select="in-scope-prefixes($x)">
           <xsl:namespace name="{.}">
```

```
              <xsl:value-of
               select="namespace-uri-for-prefix(., $x)" />
            </xsl:namespace>
         </xsl:for-each>

      <qai:Retrieve>
         <xsl:attribute name="qai:queryID">
         </xsl:attribute>
         <xsl:attribute name="dig:ontologyURI">
            <xsl:value-of select="*/owl-ql:kbRef/@rdf:resource"/>
         </xsl:attribute>
         <xsl:for-each select="owl-ql:answerSizeBound">
            <xsl:attribute name="qai:ntuples">
               <xsl:value-of select="."/>
            </xsl:attribute>
         </xsl:for-each>
         <xsl:for-each select="/*/owl-ql:serverContinuation">
<!-- Assumed as proactive value if the element exists -->
            <xsl:attribute name="mode">proactive</xsl:attribute>
         </xsl:for-each>
         <xsl:apply-templates select="owl-ql:mustBindVars"/>
         <xsl:apply-templates select="owl-ql:queryPattern"/>
         <xsl:apply-templates select="owl-ql:ServerContinuation"/>
         <xsl:apply-templates select="owl-ql:ServerTermination"/>
      </qai:Retrieve>
   </qai:RequestMessage>
  </xsl:template>

  <xsl:template match="owl-ql:mustBindVars">
     <qai:QueryHead>
        <xsl:element name="QueryVariable">
           <xsl:attribute name="URI">
              <xsl:value-of select="concat ('#',local-name(*))"/>
           </xsl:attribute>
        </xsl:element>
     </qai:QueryHead>
  </xsl:template>
```

```
<xsl:template match="owl-ql:queryPattern">
   <QueryBody>
      <xsl:for-each select="//rdf:Description[1]/@rdf:about">
         <qai:ConceptQueryAtom>
            <qai:QueryVariable>
               <xsl:attribute name="URI" >
                  <xsl:copy-of select="."/>
               </xsl:attribute>
            </qai:QueryVariable>
            <owl11xml:OWLClass>
               <xsl:attribute name="owl11xml:URI">
                  <xsl:value-of select="//rdf:type/@rdf:resource"/>
               </xsl:attribute>
            </owl11xml:OWLClass>
         </qai:ConceptQueryAtom>
      </xsl:for-each>
      <xsl:for-each select="//rdf:Description[2]/@rdf:about">
        <RoleQueryAtom>
           <qai:QueryVariable>
              <xsl:attribute name="qai:URI" >
                 <xsl:value-of select="."/>
              </xsl:attribute>
           </qai:QueryVariable>
           <qai:QueryVariable>
              <xsl:attribute name="qai:URI" >
                 <xsl:copy-of
                  select="//rdf:Description/*/@rdf:resource"/>
              </xsl:attribute>
           </qai:QueryVariable>
           <xsl:for-each select="(//rdf:RDF/rdf:Description/*)">
              <owl11xml:ObjectProperty>
                 <xsl:attribute name="owl11xml:URI" >
                    <xsl:value-of select="local-name()"/>
                 </xsl:attribute>
              </owl11xml:ObjectProperty>
           </xsl:for-each>
        </qai:RoleQueryAtom>
      </xsl:for-each>
```

```
    </qai:QueryBody>
  </xsl:template>

  <xsl:template match="//rdf:RDF/rdf:Description">
    <qai:ConceptQueryAtom>
      <qai:QueryVariable>
        <xsl:attribute name="URI" >
          <xsl:value-of select="//rdf:Description/@rdf:about"/>
        </xsl:attribute>
      </qai:QueryVariable>
      <owl11xml:OWLClass>
        <xsl:attribute name="owl11xml:URI" >
          <xsl:value-of select="//rdf:type/@rdf:resource"/>
        </xsl:attribute>
      </owl11xml:OWLClass>
    </qai:ConceptQueryAtom>
  </xsl:template>

  <xsl:template match="owl-ql:serverTermination">
    <dig:Release/>
<!-- No queryID -->
    </xsl:template>

</xsl:stylesheet>
```

B.2 Response

```
  <?xml version="1.0" encoding="iso-8859-1"?>
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   version="2.0"
   xmlns:owl-ql="http://www.w3.org/2003/10/owl-ql-syntax#"
   xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns="http://dl.kr.org/dig2.0#"
   xmlns:owl11xml="http://www.w3.org/2006/12/owl11-xml#"
   xmlns:qai="http://extension/QueryAnswers"
   exclude-result-prefixes="owl11xml qai xsl">
```

```
<xsl:output method="xml" indent="yes"  encoding="ISO-8859-1"/>

<xsl:strip-space elements="*"/>

<xsl:template match="qai:ResponseMessage">
   <xsl:apply-templates select="qai:QueryAnswers"/>
</xsl:template>

<xsl:template match="qai:QueryAnswers">
   <owl-ql:answerBundle>
      <xsl:apply-templates select="qai:Binding"/>
      <xsl:apply-templates select="qai:notifier"/>
   </owl-ql:answerBundle>
</xsl:template>

<xsl:variable name="var1" select=
"substring-after(//qai:QueryHead/qai:QueryVariable/qai:@URI,'#')"/>
<xsl:variable name="var2" select="concat('var:',$var1)"/>

<xsl:template match="qai:Binding">
  <owl-ql:answer>
   <owl-ql:binding-set>
      <xsl:element name="{$var2}">
         <xsl:attribute name="rdf:resource" >
            <xsl:value-of
             select="//owl11xml:Individual/@owl11xml:URI"/>
         </xsl:attribute>
      </xsl:element>
   </owl-ql:binding-set>
   <owl-ql:answerPatternInstance>
      <rdf:RDF
       xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#">
         <rdf:Description>
            <xsl:attribute name="rdf:about">
               <xsl:value-of select=
               "//owl11xml:Individual/@owl11xml:URI"/>
            </xsl:attribute>
```

```
            </rdf:Description>
          </rdf:RDF>
       </owl-ql:answerPatternInstance>
     </owl-ql:answer>
  </xsl:template>

  <xsl:template match="qai:notifier">
     <owl-ql:continuation>
       <xsl:choose>
         <xsl:when test="@qai:message='last'">
            <owl-ql:termination-token>
               <xsl:choose>
                  <xsl:when test="@qai:message='last'">
                    <owl-ql:end/>
                  </xsl:when>
               </xsl:choose>
            </owl-ql:termination-token>
         </xsl:when>
         <xsl:otherwise>
            <owl-ql:continuation-token>
               <owl-ql:processHandle>
                  <xsl:value-of select="//qai:QueryAnswers/@qai:asID"/>
               </owl-ql:processHandle>
               <owl-ql:answerBundleSize>
                  <xsl:value-of select="//qai:notifier/@qai:message"/>
               </owl-ql:answerBundleSize>
            </owl-ql:continuation-token>
         </xsl:otherwise>
       </xsl:choose>
     </owl-ql:continuation>
  </xsl:template>
</xsl:stylesheet>
```

Bibliography

[1] *New Racer Query Language.* http://users.encs.concordia.ca/~haarslev/racer/racer-queries.pdf.

[2] *RacerPro, an OWL reasoner and inference server for the Semantic Web.* http://www.racer-systems.com/.

[3] Alissa Kaplunova and Ralf Möller, *DIG 2.0 Proposal for a Query Interface* (2006). http://www.sts.tu-harburg.de/~al.kaplunova/dig2-query-interface.html.

[4] Alissa Kaplunova, Atila Kaya and Ralf Möller, *First Experiences with Load Balancing and Caching for Semantic Web Applications*, Institute for Software Systems (STS), Hamburg University of Technology, Germany, 2006. http://www.sts.tu-harburg.de/tech-reports/papers.html.

[5] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernàndez, Michael Kay, Jonathan Robie and Jèrôme Simèon, *XML Path Language 2.0* (2007). http://www.w3.org/TR/xpath20/.

[6] Sean Bechhofer and Boris Motik, *DIG 2.0 Specification Editor's Draft* (2006). http://www.cs.man.ac.uk/~bmotik/dig/dig_specification.html.

[7] Sean Bechhofer, *DL Interface* (2006). http://dl.kr.org/dig/interface.html.

[8] Bernardo Cuenca Grau, Boris Motik, and Peter Patel-Schneider, *OWL1.1* (2006). http://owl1_1.cs.manchester.ac.uk/xml_syntax.html.

[9] David C. Fallside and Priscilla Walmsley, *XML Schema Part 0: Primer Second Edition* (2004). http://www.w3.org/TR/xmlschema-0.

[10] *DIG 2.0 Schema.* http://www.cs.man.ac.uk/~bmotik/dig/schema/dig2.0.xsd/.

[11] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi and Peter F. Patel-Schneider (ed.), *The description logic handbook: Theory, implementation and applications*, Cambridge University Press, 2003.

[12] Alissa Kaplunova and Ralf Möller, *DIG 2.0 Concrete Domain Interface Proposal* (2006).
http://www.sts.tu-harburg.de/~al.kaplunova/dig-cd-interface.html.

[13] Michael Kay, *XSL Transformations 2.0* (2007). http://www.w3.org/TR/xslth20/.

[14] R. Fikes, P. Hayes and I. Horrocks, *OWL-QL - A Language for Deductive Query Answering on the Semantic Web*, Technical Report KSL-03-14, Knowledge Systems Lab, Stanford University, CA, USA, 2003.

[15] *Saxon 8, The XSLT and XQuery Processor*, Saxonica.
http://saxon.sourceforge.net/.

[16] Sean Bechhofer, *DIG 2.0: The DIG Description Logic Interface* (2006).
http://dig.cs.manchester.ac.uk.

[17] Thorsten Liebig, Anni-Yasmin Turhan, Olaf Noppens and Timo Weithöner, *DIG 2.0 Proposal for Accessing Told Data* (2006).
http://www.informatik.uni-ulm.de/ki/Liebig/told-access.html.

[18] Anni-Yasmin Turhan and Yusri Bong, *DIG 2.0 Proposal for The Standard Extension: Non-standard Inferences* (2006).
http://lat.inf.tu-dresden.de/~turhan/NSI.html.

[19] *XML Schema Validator tool*, DecisionSoft Limited.
http://tools.decisionsoft.com/schemaValidate.

[20] *Xalan-Java Version 2.7.0*, Apache. http://xml.apache.org/xalan-j/.

[21] *XMLBeans*, Apache. http://xmlbeans.apache.org/.