Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer gaug-zfi-bsc-2008-02

Bachelorarbeit

im Studiengang "Angewandte Informatik"

DTD-Driven Export of OWL Data to XML

Heiko Vollmann

Arbeitsgruppe für

Datenbanken und Informationssysteme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

16. Januar 2008

Georg-August-Universität Göttingen

Zentrum für Informatik

Lotzestraße 16-18

37083 Göttingen

Germany

Tel.	+49 (5 51) 39-1 44 14
Fax	+49 (5 51) 39-1 44 15
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 16. Januar 2008

Abstract

The Semantic Web will consist of a large number of autonomously evolving information systems. The processing of the data within the nodes is usually done using the RDF data model; sometimes it is based on a –specific– XML representation. These systems are able to communicate with each other, for which an XML markup is used. The common way to represent RDF data in XML is RDF/XML, which is based on an arbitrary depth-first serialization of the RDF graph. This is sufficient as long as the receiver maps the data back to RDF. In cases where a specific XML markup according to a given DTD or XML Schema is required, the serialization must be target-driven; this is e.g. the case in the MARS project for rules in the Semantic Web where certain XML markup languages are mixed. In this thesis, a tool is developed that transforms an RDF model into an XML document according to a given schema. The transformation is divided into two steps, each of which is controlled by a DTD.

Bachelor Thesis

DTD-Driven Export of OWL Data to XML

Heiko Vollmann

16. Januar 2008

supervised by Prof. Dr. Wolfgang May
Databases and Information Systems Group
Institute for Computer Science
University of Göttingen

Inhaltsverzeichnis

1 Introduction

So far, the World Wide Web has been, for the most part, a more or less static collection of web pages. These page are designed in order to be visually presented to humans. The nodes within the net that provide these pages are web servers that basically store the pages and deliver them on request. The nodes have little internal logic and almost no knowledge of the surrounding nodes in the net. Therefore they have few capabilities of interacting with other nodes.

In order to change this, the Semantic Web is being developed. The nodes in the Semantic Web are designed to interact with each other. Each node will not only be able to answer requests, but also to send requests and messages to other nodes itself.

Each node in the Semantic Web represents an autonomously evolving information system. The nodes can not only store information, but also process the data. This means that they can request more information from other nodes by themselves if their local information is not sufficient. Or they can update their data upon messages received from other nodes inside the net.

These nodes are autonomous, evolve heterogeneously and do not necessarily use a standardized internal language.

The engines, running in the web nodes, that provide the means of processing and reasoning about the data, use an RDF representation of the information to process the data and a XML representation to exchange data. Therefore, a transformation from RDF to XML is needed. An easy way to represent RDF data in XML is RDF/XML. This is sufficient for simple RDF data models, but the MARS Framework requires the RDF model to embed different languages and namespaces, therefore the RDF/XML representation is not sufficient for the MARS Framework.

This thesis provides a tool to transform RDF to XML, according to the special requirements of the MARS Framework for Evolution and Reactivity in the Semantic Web (see [17] and [18]).

The transformation process is divided into two steps.
The first step determines the language and namespace used in the RDF model and generates a XML representation. The XML is generated according to a DTD called 'striped DTD'. The resulting XML document is similar to a RDF/XML representation, but since the transformation done according to a DTD the user can determine which elements are exported to the XML document.
The second step is basically a projection from the intermediate XML document generated in step one to the final XML document. This projection is done according to an DTD called 'stripped DTD'.
Both DTD's must be given by the language designer.

The thesis is structured as follows. The next chapter will give an overview of the concepts and languages used in this thesis. Chapter 3 will then describe the technologies and APIs used to develop and implement the program. The basic ideas and concepts are presented in Chapter 4. This chapter also provides all the knowledge needed to use the tool. It outlines all the necessary and optional objects that are needed for the transformation and describes the content of the important files in detail. The implementation is described in Chapter 5. Finally, the thesis is concluded with a summary of the important parts of the transformation.

2 Basics

This thesis is developed and implemented for the use in the Framework for Evolution and Reactivity in the Semantic Web, which is introduced in [17].

This chapter will give a basic introduction to the concepts and languages, used in the design or implementation of this thesis. To give a complete overview over the concepts some related structures are described, that are not used in the thesis.

2.1 URI

A Uniform Resource Identifier[1] (URI) is a character string which uniquely identifies a resource. The resource can be electronic, like a website or located in the real world, like a person.

URIs can amongst other be classified as Uniform Resource Locators (URL) or Uniform Resource Names (URN). A URL identifies a resource through its primary access mechanism or its location within the network, for example: http://... directs to a specific webpage. A URN identifies a resource through its name in a namespace, for example isbn:0-19-853737-9 refers to a book, but provides no information where to find or get it.
A URI looks like this: *<scheme>#<scheme-specific part>*
In this case the <scheme> is the namespace or network and <scheme-specific part> is the name or location of the resource.
For more information see [7].

[1] Also called Universal Resource Identifier

2.2 XML and related structures

2.2.1 XML

The Extensible Markup Language (XML) is a meta-language, which is used to describe the structure of documents. It was derived from the SGML (see [1]) standard. XML documents are used to exchange or store data in a predefined format. Anyone who has knowledge of this predefined format can read and interpret a given document correctly. XML is used to create human and machine-readable documents.

In XML, element tags and attributes are used to define a tree structure. All real data is stored as text nodes or attribute values within the document structure. The XML markup provides the syntax of the elements, attributes and their order.

To specify the exact names of element nodes and attributes a Document Type Definition (DTD; see below) or XML Schema Definition[2] is required. The information contained in these definitions should be known to anyone using the XML document.
A short example of what an XML document could look like is as follows:

```
<root-element>
    <subelement-one attribute-one="value" attribute-two="another value" >
        text element
    </subelement-one>
    <subelement-two attribute="value" >
        another text element
    </subelement-two>
    ...
</root-element>
```

For more information see [2].

[2]XML Schema is a more advanced language to describe XML documents, but not required in this case.

2.2.2 DTD

The Document Type Definition (DTD) is a declaration for XML files and determines the structure of the XML document.

The DTD defines all names allowed for elements and attributes, and the hierarchical structure in which they may be used. It also contains information about the types of attributes and the allowed cardinality of elements and attributes.
A short example of what a DTD declaration could look as follows:

```
<!ELEMENT root-element (subelement-one, subelement-two)>

<!ELEMENT subelement-one (#PCDATA)>
<!ATTLIST subelement-one attribute-one CDATA #IMPLIED
                      attribute-two CDATA #IMPLIED>

<!ELEMENT subelement-two (#PCDATA)>
<!ATTLIST subelement-one attribute CDATA #REQUIRED
                      another-attribute CDATA #IMPLIED>
```

For more information see [3].

2.2.3 XPATH

The XML Path Language (XPath) is a simple language to access nodes, attributes or sets of these in XML documents.

XPath provides methods to navigate within a given XML document. Each navigation step in a XPath expression can contain an axis[3] on which to navigate on and a simple filter to get only nodes with specific conditions.
XPath is the base for some more advanced XML techniques such as XQuery (see blow).
For more information see [4]

[3]For example: parent, child or sibling

2.2.4 XQuery

The XML Query Language (XQuery) is a full query language for XML documents. Its syntax is similar to the commonly known SQL[4] syntax.

XQuery extends XPath and provides some programming language features, such as variable binding, loops, complex conditioning and creation of complex return sets.
For more information see [5].

2.2.5 XSLT

The Extensible Stylesheet Language Transformations (XSLT) is a language used to transform XML documents. The result document of the transformation can have a structure that is completely different from the structure of the source document. A Stylesheet can be used, for example, to transform a given XML document into one that is valid for another DTD.

The transformation rules of the XSLT are applied recursively to the elements of the XML document.
For more information see [6].

2.3 Ontology

In computer science the term *ontology* specifies a set of definitions which describe the concepts of a domain. The ontology contains information about the hierarchical structure of the classes in the domain, the notation used and the semantics which lie behind the notation.

The ontology defines the objects which the domain can contain and specifies the relations these objects are linked to each other with. It determines the exact notation to be used in any document about the specified domain.
For more information see [19]

[4]Structured Query Language, known from relational databases

2.4 RDF and RDF Schema

2.4.1 RDF

The Resource Description Framework (RDF) is a structure to describe resources and denote this description in a machine readable and processable way.

RDF uses URIs (see above) to uniquely identify a resource. Because of this, the resource can be located anywhere within the net or the real world. One resource can also be described by many RDF models located in different places within the net. Through the use of URIs, the distributed information about one resource can be brought together and the user can tell that the different RDF models refer to the same resource.

The RDF Model is composed of Triplets. The Triplets contain:

- a *subject* which always is the URI of the resource, this triple gives information about,
- a *predicate* which names the aspect of the resource to be described in this triplet,
- an *object* which is the value of the description. This can be a literal (string or number) or another resource.

An RDF Model can be represented for example as a graph, a list of triples or a special XML markup called RDF/XML.
The following is an examples which illustrates the basic idea of what an RDF triple could look like in N-triplets:

```
<http://www.myworld.de/country/D> <http://www.myworld.de/meta/hasCity> <http://www
    .myworld.de/city/Goettingen> .
<http://www.myworld.de/country/D> <http://www.myworld.de/meta/name> "Germany" .
```

This gives a description about a resource which can be accessed by the URI http://www.myworld.de/country/D. It contains the information that this resource has a *hasCity* relation to another resource found through http://www.myworld.de/city/Goettingen and a *name* property which is the literal 'Germany'.

The same example denoted in RDF/XML locks like this:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns="http://www.myworld.de/meta/#">
   <rdf:Description rdf:about="http://www.myworld.de/country/D">
      <hasCity rdf:resource="http://www.myworld.de/city/Goettingen"/>
      <name>Germany</name>
   </rdf:Description>
</rdf:RDF>
```

For more information see [9].

2.4.2 RDFS

RDF Schema (RDFS) is a vocabulary description language.
Via an RDF Schema the vocabulary of a specific domain is defined. A Schema provides meta data to the data contained in the RDF model. The meta data can for example contain class declarations and information about the hierarchy of the classes (*rdfs:subClassOf*). Furthermore it can contain domain and range information about the properties used in the RDF model. Simple ontologies can be represented in RDFS.

Syntactically, an RDFS document is a valid RDF Model and can be read and processed with the same mechanisms as RDF.
For more information see [10].

2.4.3 SPARQL

SPARQL is a recursive acronym that stands for SPARQL Protocol And RDF Query Language.

SPARQL is a query language used to access RDF graphs. Similar to XQuery (see above), SPARQL syntax is based on known SQL commands, but also allows variable bindings and optional parameters.

To make the queries easier to read and write for humans SPARQL provides methods for namespace handling: In the beginning of a query the 'PREFIX' keyword can be given to define a set of namespaces for the query. The query itself can then use briefer names, called local names.
This is a short query, to get a basic idea, which illustrates what a SPARQL query could look like:

```
PREFIX : <http://www.semwebtech.de/mondial/10/meta#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?names
   {
   ?country rdf:type :Country.
   ?country :name ?names
   }
```

Explanation: The query will search the RDF graph for all resources which have a property 'rdf:type' and ':Country' as value of this property. For all these resources it selects the value of the ':name' property and returns these values in the names field.
For more information see [11].

Since the expressive power of RDFS is not sufficient to express all ontology features needed in the Semantic Web and since this thesis is developed to be used in the Semantic Web, another Language is needed. This is described in the next chapter.

2.5 OWL

The Web Ontology Language (OWL) is a language to define and instantiate ontologies. It is technically based on RDF syntax, but extends RDF and RDFS by the possibilities of expressing semantics related to the data.

OWL is designed to note down the structure and semantics defined by the theoretical

construct of an ontology, needed for the Semantic Web.
This is a small example of a relation needed for reasoning that can not be expressed in RDFS: *hasCity* is the inverse of *cityIn*.
In OWL syntax it looks like this :hasCity owl:inverseOf :cityIn.
For more information see [8].

3 Software tools

3.1 The JENA Framework

Jena is an open source framework developed to provide structures and methods needed to build Semantic Web applications. It has developed out of the work on the HP Labs Semantic Web Programme [16]. It provides methods to handle RDF, RDFS, OWL and SPARQL.

3.1.1 Jena API

The Jena API provides methods to handle RDF data.
The main purpose of the Jena API is to store and access an RDF model with Java Classes. It provides methods to load RDF data from one or more files and builds an RDF model from the data. This model can then be accessed and altered through numerous functions in the API. Simple but limited query and navigation methods are provided for the RDF graph, as well as export functions, for example to save the graph in an RDF/XML markup.

Since the build-in navigation function is limited to simple queries to get all nodes with a special property or all properties of a given node, the Jena API provides an engine to process queries written in common query languages, such as SPARQL. This engine is called ARQ[1].

[1]Since ARQ is not important for this thesis, no further details are given.

3.2 Reasoner

Through a reasoner, an API can draw conclusions about the RDF graph. The meta information can be given in RDFS or OWL code terms within the model. All queries posed or commands given to the RDF model will be processed by the reasoner.

The Jena API also provides methods for loading a reasoner. The reasoner can either be one of those provided by the API or it can be an external one, as for example the 'Pellet Reasoner' (see below).

To give a better understanding of this, consider this example:
Assume that in the RDF model there are *Country* and *City* objects. Each *City* shall have a *:cityIn* property which directs to the *Country* where the *City* is located. The *Country* shall have no information about what *City* lies in it.

Assume further that the OWL meta data in the RDF model contains the following entry: :hasCity owl:inverseOf :cityIn. Now consider sending the following SPARQL request to the model:

```
SELECT ?city
WHERE {http://www.myworld.de/countries/D/ :hasCity ?city}
```

Because the reasoner applies the information that *:hasCity* is the inverse of *cityIn* to the model, the answer of this query will be all cities that are located in the country http://www.myworld.de/countries/D/
Without the reasoning the answer would be empty, since the *Country* objects do not actually have a property *:hasCity*.

3.2.1 Pellet reasoner

The Pellet reasoner was originally developed at the University of Maryland's Mindswap Lab. It is designed to provide the full expressivity of OWL-DL for Java applications.
The reasoner can be integrated directly into the Jena model creation process.
For more information see [15].

3.3 LOG4J

Log4j is a framework for logging messages in Java applications. It is part of the 'Apache Software Foundation'. Its main purpose is to provide a simple-to-use method for logging during development, debugging and deployment of an application.

Log4j provides six basic logging levels: FATAL, ERROR, WARN, INFO, DEBUG and TRACE. Theses logging levels are hierarchical. In this case this means that, if the log level is set to WARN all WARN, ERROR and FATAL messages will be logged.

The configuration for Log4j is given in a separate file[2]. Therefore it is not necessary to edit the source code in order to change the log level, the output device (stdout, file, etc.) or the logging format.
For more information see [14].

3.4 DTDParser

The DTDParser API[3] was created by Mark Wutka (see [12]) and deployed under LGPL[4]. It is designed to parse a DTD into Java objects.

The API provides methods to parse a DTD file into an *com.wutka.dtd.DTD* object from a URI- or Readerobject, therefore a DTD can be obtained from anywhere within in the net, the local file system or an application. All elements, attributes, their types and cardinalities are wrapped into objects and can be accessed through methods build within the API. The hierarchical structure of the classes of the API reproduces the structure of the DTD file so that it is possible to navigate through the tree of elements and attributes. This can be used to iterate though the DTD and create an XML instance valid to the given DTD, as done in this thesis.
For more information and a complete javaDoc see [13].

[2]Filename: log4j.properties
[3]Application Programming Interface
[4]GNU Lesser General Public License

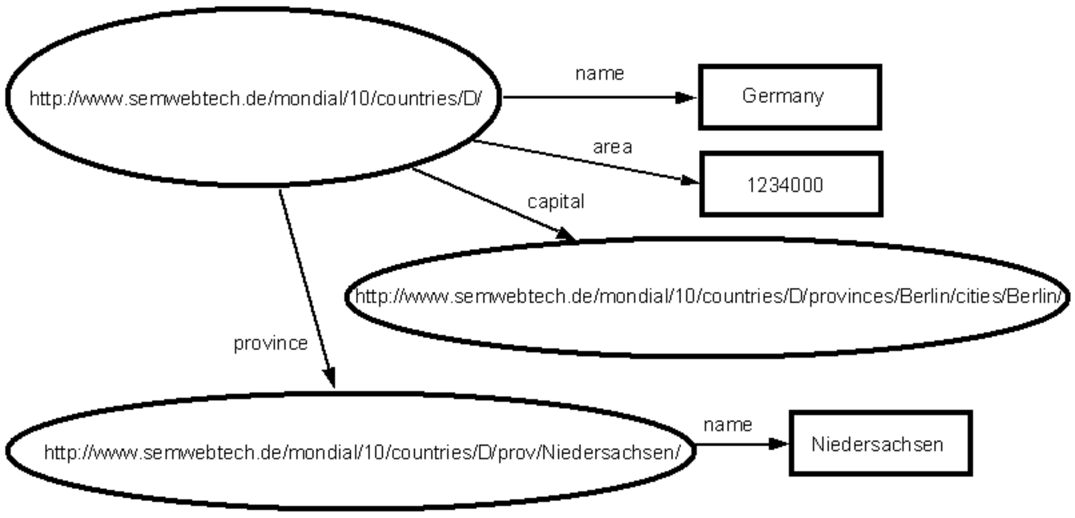
4 Design

In this Chapter the basic design of the transformation tool is presented and the specifications of the required parts are explained.
In the first chapter an example of all input and output documents is given.
The following chapters will describe the concepts of this thesis and explain the in- and output documents in detail. Beginning with a general introduction in chapter two. The next chapters three, four and five describe the components of the DTD's and its functions in detail. Chapter six describes the handling of ID and IDREF elements. In Chapter seven the change of the language is described. Finally Chapter eight describes the use of Log4J and tis use to debug the DTD's.

4.1 Basic example

This is a complete example of the in- and output of the transformation tool. The following chapters will explain the important parts in detail, therefore no further explanation is given here.

RDF model: This is the RDF model.


http://www.semwebtech.de/mondial/10/countries/D/
name
Germany
area
1234000
capital
http://www.semwebtech.de/mondial/10/countries/D/provinces/Berlin/cities/Berlin/
province
http://www.semwebtech.de/mondial/10/countries/D/prov/Niedersachsen/
name
Niedersachsen

Striped DTD: This is the striped DTD, which will be used in the first step. For details see chapter 4.4.

```
<!ELEMENT Country ( name, area, capital, hasProvince)>
<!ATTLIST Country rdf:about CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT area (#PCDATA)>
<!ELEMENT capital EMPTY>
<!ATTLIST capital rdf:resource CDATA #REQUIRED>
<!ELEMENT hasProvince (Province)>
<!ELEMENT Province (name)>
```

```
<!ATTLIST Province rdf:about CDATA #REQUIRED>
```

Stripped DTD: This is the stripped DTD, which will be used in the second step. For details see chapter 4.5.

```
<!ELEMENT Country (name, Province*)>
<!ATTLIST Country capital IDREF #IMPLIED
                  area CDATA #IMPLIED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT Province (name)>
```

Result: This is the resulting XML document.

```
<Country area="1234000" capital="http://www.semwebtech.de/.../cities/Berlin/">
   <name>Germany</name>
   <Province>
      <name>Niedersachen</name>
   </Province>
</Country>
```

4.2 General transformation

As mentioned in the introduction the MARS framework requires a method to control the details of the transformation from an RDF model to an XML document. In order to give the user the opportunity to control the transformation process two DTD's are used.

The transformation is dived into two general steps. For each step a DTD is needed to determine the contents of the document generated in this step. The first step creates a intermediate RDF/XML document. The DTD used for this step is called 'striped DTD'. The second step is a projection form the intermediate RDF/XML to the final XML document, according to the DTD called 'stripped DTD'.
The DTDs are described in detail in the sections 4.4 and 4.5.

First Step

In the first step, an RDF/XML document is generated to represent the RDF model.

This step creates the elements in the XML document according to the structure specified by the striped DTD (see below). The objects referred to by the references in the RDF model are created as elements. Normally, the class of the object is taken to be the name of the XML element, super classes of the objects are also allowed.

The properties of the objects are used to create child elements of the objects. The *predicate* will be the name of the child element and the *object* will be the value of the element or the attribute (for details, see below).

All the attributes created in this step are used to indicate what kind of element shall be created for the final XML document and to store the URIs of objects and relations needed in the second step. This also means that there are no attributes allowed in the striped DTD, except for those named in section 4.4 and 4.5.

Second Step

The second step transforms the RDF/XML document created in step one to the final XML document. In order to accomplish this, the stripped DTD is used.

Since the basic elements are created in step one, this steps is a projection from the RD-F/XML createt in step one to the final XML document. The ID and IDRDF and all other attributes are created in this step. The ID creation is described in section 4.6. The other attributes are created from the child elements, created in step one.

4.3 General DTD

The striped and stripped DTD are the central inputs in the transformation process. They contain the directions concerning how to transform the RDF nodes and properties into XML elements and attributes. It is very important for both DTDs that the names given to the elements and attributes exactly correspond to the names in the RDF model.

The locations of the DTDs have to be given in the language definition and accociate to the MARS Framework. This is an example of how the beginning of a language definition could look like:

```
<mars:ECALanguage rdf:about="http://www.semwebtech.org/languages/2006/eca-ml#">
 <mars:shortname>eca-ml</mars:shortname>
 <mars:name>ECA Markup Language</mars:name>
 <mars:hasDTD rdf:resource="http://www.semwebtech.org/languages/2006/ecaml.dtd"/>
 <mars:hasStripedDTD
   rdf:resource="http://www.semwebtech.org/languages/2006/ecaml-striped.dtd"/>
   ...
 <mars:ECALanguage>
```

As can be seen in this example, the language definition has two properties for the DTDs. One is the *hasStripedDTD* to provide the striped DTD and one *hasDTD* to provide the stripped DTD. Both properties need to have a URI directing to the actual DTDs.

The language definitions are part of the metadata within the RDF model. Therefore, the transformation tool will get the DTD URIs from the metadata and will load the DTD from the locations given in the URIs. Whenever the language is changed during the transforming process, new DTDs for the new language are loaded through the method described above.

4.4 Striped DTD

The striped DTD is the base for the first step of the transformation.
The striped XML document, corresponding to the striped DTD, is the intermediate step in the transformation. The elements need to correspond to the objects given in the RDF graph. Therefore no free attributes are allowed but some special designed attributes must be given either to indicate the type of element to be generated in the XML document or to pass basic information to the second step of the transformation.

There are four basic cases that are possible in the transformation from an RDF model of any form to an RDF/XML document. The possibilities arise from the two possible types of input and the two possible types of output. The input coming from the RDF model can either be a resource or a literal. The output to the RDF/XML document can either be an element or an attribute. Actually there is a fifth case, which derives from the case of an *object* in an RDF triple being a link to an actual resource and the need of creating an XML element from this resource.

1. Building an XML element from an RDF resource.
   This is the circumstance to build an XML element according to a Node in the RDF graph. The element node will be created and an 'rdf:about' attribute will be given to the element.
   The element node can either have children or be empty.
   There has to be one attribute whose name has to be *rdf:about* and its type has to be *CDATA*. No other attributes are allowed here.
   These are two short examples of what a DTD entry for this kind of node could look like:
   RDF

http://www.semwebtech.de/mondial/10/countries/D/

   striped DTD

```
<!ELEMENT Country (capital, area, name, hasProvince)>
<!ATTLIST Country rdf:about CDATA #REQUIRED>
```

   XML/RDF

```
<Country rdf:about="http://www.semwebtech.de/mondial/10/countries/D/" />
```

2. Building an attribute from an RDF resource.

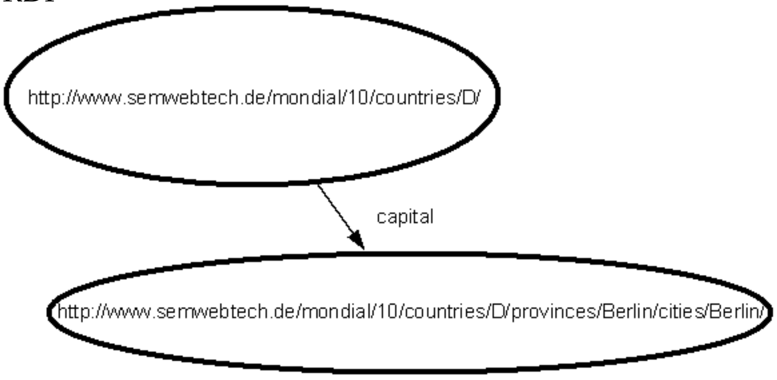
   This is the circumstance to build an XML *IDREF* or *IDREFS* attribute in the XML document. To find the element referred to in the second step of the transformation, the *rdf:resource* string must be given to the element.

   The element node has to be empty.

   There has to be one attribute whose name has to be *rdf:resource* and its type has to be *CDATA*. No other attributes are allowed here.

   This is a short example of what a DTD entry for this kind of Node could look like:

   RDF


http://www.semwebtech.de/mondial/10/countries/D/
capital
http://www.semwebtech.de/mondial/10/countries/D/provinces/Berlin/cities/Berlin/

   striped DTD

```
<!ELEMENT Country (capital)>
<!ELEMENT capital EMPTY>
<!ATTLIST capital rdf:resource CDATA #REQUIRED>
```

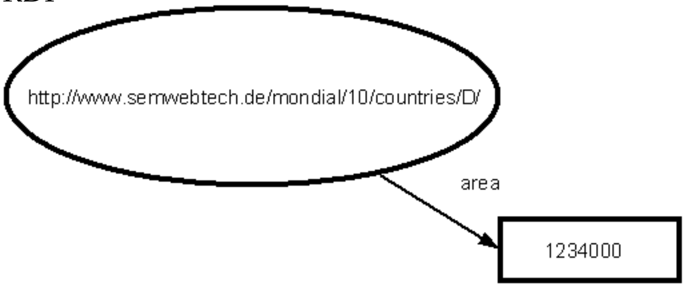
   XML/RDF

```
<Country>
   <capital rdf:resource="http://www.semwebtech.de/.../cities/Berlin/">
</Country>
```

3. Creating an attribute from a literal.

   In this case a simple XML element is created from an RDF literal. The information that this element node will later on evolve to an attribute is to be stored in the striped

DTD.
The element node has to have a single #*PCDATA* entry.
No attributes are allowed here.
This is a short example of what a DTD entry for this kind of Node could look like:
RDF



striped DTD

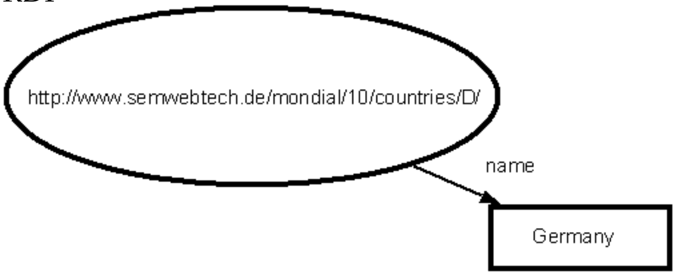
```
<!ELEMENT Country (area)>
<!ELEMENT area (#PCDATA)>
```

XML/RDF

```
<Country>
    <area>1234000</area>
</Country>
```

4. Building an XML Element from a literal.
   Similar to case three, a simple XML Element is created in this case.
   The Element node has to have a single #*PCDATA* entry.
   No attributes are allowed here.
   This is a short example of what a DTD entry for this kind of Node could look like:

RDF



striped DTD

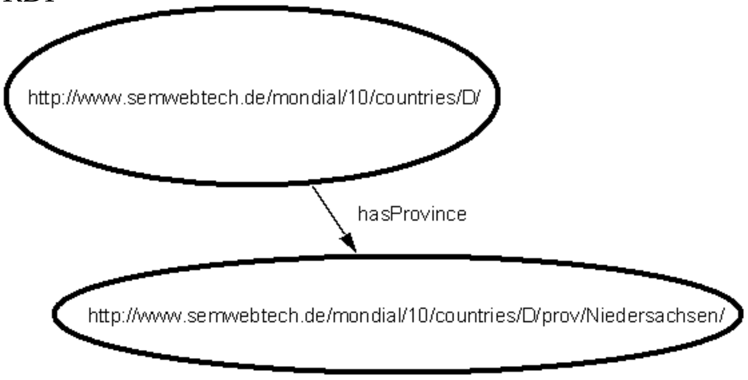
```
<!ELEMENT Country (name)>
<!ELEMENT name (#PCDATA)>
```

XML/RDF

```
<Country>
   <name>Germany</name>
</Country>
```

5. Building the referred XML element from an RDF property and object.
   This case represents the possibility of having a reference as an *object* in the RDF triple and the need to generate the object referred to as an XML element. In contrast to case one an XML element is created from the predicate of the RDF triple.
   The element node has to have a single child entry with the name of the element to be created for the final XML document.
   No attributes are allowed here.
   There has to be a case one element in the DTD to define what the child of this node should look like.
   This is a short example of what a DTD entry for this kind of node could look like:

RDF


http://www.semwebtech.de/mondial/10/countries/D/
hasProvince
http://www.semwebtech.de/mondial/10/countries/D/prov/Niedersachsen/

striped DTD

```
<!ELEMENT Country (hasProvince)>
<!ELEMENT hasProvince (Province)>
<!ELEMENT Province (name)>
<!ATTLIST Province rdf:about CDATA #REQUIRED>

<Country>
    <hasProvince>
      <Province rdf:about="http://www.semwebtech.de/.../prov/Niedersachsen/">
    </hasProvince>
</Country>
```

4.5 Stripped DTD

The stripped DTD is the base for the second step of the transformation and the DTD for the final XML document. All elements and attributes are allowed in the stripped DTD, but it is very important that these correspond to the design of the striped DTD.

Corresponding to the five cases in the striped DTD, there are five cases in the stripped DTD as well.

1. Building an XML element from an RDF resource.
   Since the element itself already exists in the striped DTD, the only thing to be done here is to leave out the *rdf:about* attribute, which is not needed in the final XML document.
   RDF/XML
   ```
   <Country rdf:about="http://www.semwebtech.de/mondial/10/countries/D/" />
   ```
   stripped DTD
   ```
   <!ELEMENT Country (name, Province*)>
   ```
   final XML
   ```
   <Country/>
   ```

2. Building an attribute from an RDF resource.
   In this case an *IDREF* attribute is generated from the element generated in the first step. (See section 4.6 for details.)
   The given attribute has to be of type *IDREF* or *IDREFS*[1], otherwise, the URI stored in the *rdf:resource* attribute in the first step will be the value of a normal attribute.
   RDF/XML
   ```
   <Country>
      <capital rdf:resource="http://www.semwebtech.de/.../cities/Berlin/">
   </Country>
   ```
   stripped DTD
   ```
   <!ELEMENT Country (name, Province*)>
   <!ATTLIST Country capital IDREF #IMPLIED>
   ```
   final XML
   ```
   <Country capital="http://www.semwebtech.de/.../cities/Berlin/"/>
   ```

[1]IDREFS if there is more than one element to be referred to.

3. Creating an attribute from a literal.
   Similar to case two, an attribute is generated. The value of the attribute will be the contents of the element generated in step one.
   The attribute has to be of type *CDATA*.
   RDF/XML

   ```
   <Country>
       <area>1234000</area>
   </Country>
   ```

   stripped DTD

   ```
   <!ELEMENT Country (name, Province*)>
   <!ATTLIST Country area CDATA #IMPLIED>
   ```

   final XML

   ```
   <Country area="1234000" />
   ```

4. Building an XML element from a literal.
   Since the simple element was created in step one, nothing else is done in this step.
   The Element has to have a *#PCDATA* text node.
   RDF/XML

   ```
   <Country>
       <name>Germany</name>
   </Country>
   ```

   stripped DTD

   ```
   <!ELEMENT Country (name, Province*)>
   <!ELEMENT name (#PCDATA)>
   ```

final XML

```
<Country>
    <name>Germany</name>
</Country>
```

5. Building the referred XML element from an RDF resource
   The intermediate element that represents the link to the actual object may be deleted for the final version of the XML document.
   RDF/XML

```
<Country>
    <hasProvince>
      <Province rdf:about="http://www.semwebtech.de/.../prov/Niedersachsen/">
    </hasProvince>
</Country>
```

stripped DTD

```
<!ELEMENT Country (name, Province*)>
<!ELEMENT Province (name)>
```

final XML

```
<Country>
    <Province \>
</Country>
```

4.6 ID and IDREF

[2] The basic structure of an RDF model is a graph, which means that there are cycles possible within it.
To give a basic idea of what such a cycle could look like, this is a short example:

[2]Wherever IDREF is written in this chapter, IDREFS are allowed to.

```
<http://www.family.de/person/Anton><http://www.family.de/meta/hasChild><http://www
    .family.de/person/Hans> .
<http://www.family.de/person/Anton><http://www.family.de/meta/hasWhife><http://www
    .family.de/person/Susanne>
<http://www.family.de/person/Susanne><http://www.family.de/meta/hasChild><http://
    www.family.de/person/Hans> .
```

This example indicates that *Anton* and *Susanne* both have a child and the child is the same (it has the same URI).

The basic structure of XML is a nested tree. Therefore it is not possible to represent a loop in XML. In the example given above, the same child is needed two times, once as a subelement of Anton and once as a subelement of Susanne.

This problem can only be solved with IDs and IDREFs. By the use of these, the three persons can be created as individual elements and then linked through IDs and IDREFs.

```
<Person hasChild='pId0001' hasWhife='pId0002'>
    <name>Anton</name>
</Person>
<Person hasChild='pId0001' id='pId0002'>
    <name>Susanne</name>
</Person>
<Person id='pId0001'>
    <name>Hans</name>
</Person>
```

In this example *hasChild* is of the type IDREF and *id* is of the type ID. The types of the attributes are indicated in the corresponding DTD.

The instruction to generate an ID needs to be encoded in the stripped DTD, since there are no free attributes allowed in the striped DTD. There are two possible ways to generate

IDs in the transformation.

One way is to explicitly provide the id in the RDF model. To do so, there has to be a case two-element[3] in the striped DTD with the same name as the ID element and, of course, the ID element in the stripped DTD. The transformation will generate an ID element with the value of the object of the RDF model.

The second way to generate an ID is to simply leave out the element in the striped DTD and provide it only for the stripped DTD. The transformation will then generate the ID with the resource-string of the node as value for the ID element.

An IDRDF attribute is automatically created if a case-two-transformation is given in the DTDs and the attribute in the stripped DTD is given the type IDREF.

4.7 Change language

The ECA language defined in the General Framework for Evolution and Reactivity in the Semantic Web [17] can and needs to include other languages.
This example is meant to illustrate the point: An RDF model is given, following the ECA language. In this model an *Event* node exists. The properties of this *Event* node define what to do if this event occurs. If this event occurs, the snoop part of the engine should handle this event. The snoop engine needs specific information to process the occurring event. Which information needs to be given to the snoop engine is defined in a special snoop-language as part of the Framework mentioned above. In order to provide the information the properties of the *Event* node need to follow this snoop-language and the *Event* node itself needs to contain the information that from here on the used language is snoop and not eca.

Since the different languages are contained within the RDF model, the transformation tool can detect the change of language and export the subgraph with another language and its assigned DTDs. To do so, the tool requires a special design of the striped DTD and

[3]see chapter about striped DTD

the *Event* node.

The *Event* element in the striped DTD needs to contain the keyword *ANY-other-language* as a child element. It may contain other child elements as a alternatives. If other elements are given, the *ANY-other-language* element needs to be the last one in the list. Thus, the language change only occurs when none of the other elements is found in the RDF model.

For consistency the DTD needs to define an *ANY-other-language* element. This element will never be created, since in this case the tool changes the language and creates the elements of the new language as child elements of the *Event* node. This is a short example what a striped DTD entry for language switching could look like.

```
<!ELEMENT Query (Atomic | Opaque | ANY-other-language)>
<!ELEMENT Atomic (ANY-other-language)>
<!ELEMENT Opaque (#PCDATA | ANY-other-language)>
<!ELEMENT ANY-other-language ANY>
```

The RDF *Rule* node needs to have two *rdf:type* predicates. The objects of the properties need to refer to the language class nodes. Theses nodes need to have am *rdfs:definedBy* predicate. The object of this property needs to be the language description node. The transformation tool will then check both languages and switch to the language which is not in place at this point.

The following is a short example of what an *Event* node for language switching could look like:

```
<...#Rule/Query1> rdf:type <eca:Query>
<...#Rule/Query2> rdf:type <owlq:Query>
 <eca:Query> rdfs:definedBy <http://www.semwebtech.org/languages/2006/eca-ml#>
 <owlq:Query> rdfs:definedBy <http://www.semwebtech.org/languages/2006/owlq#>
```

Assuming that the language is *eca-ml* when the node is reached, the transformation tool will continue to expand the children of *Rule* with the owlq language.

4.8 Log4J and DTD debugging

As described above, it is very important that the DTDs match a very specific design. The design of the DTDs depends on the design of the language used for the RDF model. Because of this, the DTDs must be provided by the language designer, along with the language for the RDF model.

To help the language designer in the process of designing the DTDs for his language, the transformation tool provides a logging system. The logging is based on *log4J* (See section 3.2). There are two special log levels used to provide the different needs for development and deployment.

The first log level is the INFO level. This level is meant to be active during development. It provides the developer with lots of information about what the transformer is actually doing. At this level, the first step of the transformation logs every single decision which element or attribute to create. Whenever an element or attribute is created, its name is logged along with its value, if the value exists. The second step of the transformation logs every deletion of intermediate elements resulting from case five elements (see above). It also logs all created IDs and IDREF attributes along with its value, so that the developer as well can see if the ID is created from a given value or resource.

The second log level is the WARN level. This level is meant to be active during deployment. It logs entries only when an occasion appears that could result in an incorrect XML document. Such an occasion occurs, for example, if one of the DTDs is not found, or the striped DTD does not correspond to the RDF model. The latter may be the case, for example, if the DTD designates the cardinality of an element to be one-to-many, but no matching node is found in the RDF model.

All events mentioned above are written in an single row in the log file. The information is presented in the form of a sentence or part of a sentence in the log file. The developer of the language can check the log file to see if all components of the RDF model are correctly transferred to the XML document. If this is not the case, he should be able to find the part of the DTD which needs to be changed quickly.

5 Implementation

In this chapter the implementation of the transformation tool and its interface is described.

5.1 General

The tool is implemented in Java. This is required to use this tool in the Framework for Evolution and Reactivity in the Semantic Web.

All methods are within only one class. During the construction of the class, all necessary data is initialized. The required data is extracted from the language description which is within the RDF model and the required DTD's are loaded into DTDParser objects.

The DTDs are loaded from the URIs, parsed by the DTDParser (see section 3.4) and stored in objects. The DTD root element corresponding to the node where to start the transformation process is determined and stored.
The short name of the language is acquired from the language description and the prefixes are extracted and stored for later use.

The actual transformation process is sequential and divided into two steps as described in the General transformation chapter (see 4.2).
The methods implementing step one determine which nodes to expand and what XML elements to create according to the striped DTD. These elements are then created and all their children are expanded recursively. If a language change is indicated (as described in chapter 4.7), the methods execute the language change as described below.

Step two consists of three major parts, each part is implemented in a method:
The first part detects all *ID* elements in the stripped DTD and creates theses elements from the rdf:about attribute or a given node, as described in ID and IDREF (4.6).
The second step iterates through all *IDREF* attributes and creates these from the child element with the matching name.
In part three, two things happen. All other attributes are created according to the stripped DTD from the elements created in step one. This part is also responsible for deleting all the intermediate elements that are created in step one but are not required to be in the resulting XML document. This part of step two is actually the only part in the tool that iterates through the XML tree and not through the elements given in the DTDs.

5.2 Interface and internal methods

The tool has one constructor and only one public function.
The prototype of the constructor looks as follows:
RdfToXml(OntModel model, Resource node, Resource language);
The OntModel is the full RDF model, including all necessary language descriptions and all meta information. The model should be generated with a reasoner, so that all the necessary RDFS and OWL meta information can be applied correctly to the model.
The first resource is the node which is the starting point of the transformation. All sub nodes of this node according to the DTD will be in the resulting XML document.
The second resource is the node which describes the language to be used in the transformation of this part of the model.
The only public method is called generateXML(). This method starts the transformation itself. It returns the result of the transformation as a *w3c.org.dom.Document*.
All other methods are private and for internal use only.

5.3 Change language

As described in the Change language section 4.7 the tool needs to be able to change the used language in the process. This is done by recursion. If the tool encounters a node where a language change is required, it creates a new instance of the RdfToXml class and initiates this instance with the model, the actual node and the new language. The new class is then commanded to generate the subtree and the result is stored along with the actual node. After step two is finished, the result is simply plugged in at the position of the node.

The temporal storage is necessary for step two to work correctly. If the result were plugged in directly, the second step would delete all the sub nodes of the node where the language change occurred, since the DTD for the actual language does not contain the elements of the new language.

6 Conclusion

In this thesis a transformation tool has been developed that can be used to transform an RDF model into an XML document. The transformation is done according to two DTDs. These DTDs are embedded into the language description and therefore must be provided by the language designer.

The transformation itself is divided into two steps, each according to one DTD.

The tool is implemented as a Java class and therefore can be used by other applications dealing with RDF models and the need to save data as an XML document. It is designed and implemented to be used in the Framework for Evolution and Reactivity in the Semantic Web. Since this Framework has some special requirements to a transformation tool and provides some specific nodes in the RDF graph, the tool can not transform all RDF graphs to an XML document, but only those, which provide the required language nodes and the two DTD's.

The design of the Framework mentioned above is still under development. Since the tool is specific to this design, it may be necessary to adapt the tool to new specification of the framework in the future.

Literaturverzeichnis

[1] Charles F. Goldfarb *The SGML Handbook.* ISBN 0-19-853737-9. 1991. Clarendon Press

[2] Extensible Markup Language (XML): http://www.w3.org/XML/

[3] Document Type Definition (DTD): http://www.w3.org/TR/xhtml1/dtds.html

[4] XML Path Language (XPath): http://www.w3.org/TR/xpath

[5] XML Query Language (XQuery): http://www.w3.org/TR/xquery/

[6] XSL Transformations (XSLT): http://www.w3.org/TR/xslt

[7] Naming and Addressing: URIs, URLs, ... http://www.w3.org/Addressing/

[8] Web Ontology Language (OWL): http://www.w3.org/TR/owl-features/

[9] Resource Description Framework (RDF) http://www.w3.org/RDF/ :

[10] RDF Schema (RDFS): http://www.w3.org/TR/rdf-schema/

[11] SPARQL Query Language for RDF: http://www.w3.org/TR/rdf-sparql-query/

[12] Wutka Consulting, Inc.: http://www.wutka.com/index.html

[13] Wutka DTDParser: http://www.wutka.com/dtdparserdoc.html

[14] Apache log4j: http://logging.apache.org/log4j/

[15] Pellet: http://pellet.owldl.com/

[16] HP Labs Semantic Web Researc: http://www.hpl.hp.com/semweb/

[17] Wolfgang May, José Júlio Alferes, Ricardo Amador. An Ontology- and Resources-Based Approach to the Evolution and Reactivity in the Semantic Web. In Ontologies, Databases and Semantics (ODBASE), numer 3761, pages 1553-1570. Springer, 2005

[18] Reasoning on the Web with Rules and Semantics (REWERSE) http://rewerse.net/

[19] Thomas R. Gruber. *A Translation Approach to Portable Ontology Specifications.* Knowledge Acquisition, 5(2):199-220, 1993.