# A Prolog-based Approach for Analyzing XML Documents and Their Structure

Maria Salo

University of Tampere
School of Information Sciences
Computer Science
M.Sc. Thesis
Supervisor: Timo Niemi
November 2011

University of Tampere
School of Information Sciences
Computer Science
Maria Salo: A Prolog-based Approach for Analyzing XML Documents and Their Structure
M.Sc. Thesis, 48 pages + 7 appendices pages
November 2011

---

Since the first version of XML was defined in 1998, it has become the most common tool for sharing and transferring data between applications in the Internet. It is also becoming more and more popular as a means to store and describe data. However, it seems there is a need for a tool that can help those people that have to query large and mostly unknown XML documents. Traditional query languages such as XPath and XQuery require the user to know – at least to some extent – the structure of the document they are handling.

This thesis introduces a Prolog-based approach for analyzing XML documents. It features predicates with which the user can analyze the structure and content of XML documents without any previous knowledge of their purpose. The prototype described here uses the XML relation as a foundation. With a tool such as the one introduced here the user can easily check if a document is of interest to them as well as use the tool together with traditional path-oriented query languages.

Keywords and phrases: XML, Prolog, XML relation, XML analysis, dataspaces, XML-based dataspacing

## Table of Contents

# 1. Introduction

XML (Extensible Markup Language) [Bray et al., 1998] is a markup language designed to describe the structure and content of documents. It is similar to HTML (HyperText Markup Language) [Raggett et al., 1999] except that with HTML the purpose is to support the displaying of the content of the document, whereas XML is used to support the semantic interpretation of the document. Since the first version of XML was defined in 1998, it has become the most common tool for sharing and transferring data between applications in the Internet. It is also becoming more and more popular as a means to store and describe data.

Corporations need to transfer large amounts of data with their partners in order to conduct business. Especially large companies, which can have millions of data transfers a day, realized long ago that transferring the data electronically is not only faster but also more reliable, secure and traceable than doing the same data transfers by phone or fax. [Bussler, 2001] For example, transportation companies need to let their partners know of new transports that require pickup or transports that are coming into a terminal.

There are several standards that were developed for Business-to-Business (B2B) data exchange. One of the widely used ones is EDI (Electronic Data Interchange). However, because of its syntax, an EDI message is not very readable (see Illustration 1). This is one of the reasons why XML is starting to become popular also in B2B applications. XML is much more readable than EDI and it is also easily manipulated to other forms. Most businesses can handle XML documents whereas EDI messages always require changes in a company's systems. XML also provides easy definition and implementation of documents and messages exchanged over the Internet. [Bussler, 2001]

| | |
|---|---|
| UNH+ME000001+IFTMIN:D:01B:UN:EAN004' | Message header |
| BGM+610+569952+9' | Transport instruction number |
| DTM+137:20020301:102' | Message date/time 1st March 2002 |
| DTM+2:200203081100:203' | Delivery date/time requested, 8th March 2002 at 11:00 |
| CNT+11:1' | Total number of packages 1 |
| RFF+CU:TI1284' | Consignor's reference number TI1284 |
| TDT+20++30+31' | Details of transport, by truck |
| DTM+133:200203051100:203' | Estimated departure of truck 5th March 2002 at 11am |
| LOC+9+5412345678908::9' | Place of truck loading identified with GLN 5412345678908 |
| NAD+CZ+5412345123453::9' | Consignor identified with GLN 5412345123453 |
| NAD+CA+5411234512309::9' | Carrier identified with GLN 5411234512309 |
| NAD+CN+5411234444402::9' | Consignee identified with GLN 5411234444402 |
| NAD+DP+5412345145660::9' | Delivery party identified with GLN 5412345145660 |
| GID+1+1:09::9+14:PK' | First occurrence of goods in one returnable pallet with 14 packages |
| HAN+EAT::9' | The goods are foods stuffs |
| TMP+2+000:CEL' | Transport temperature 0 degrees Celsius |
| RNG+5+CEL:-5:5' | The range of temperature must be between -5 and 5 degrees Celsius |
| MOA+44:45000:EUR' | Declared valued of the carriage 45.000 EUR |
| PIA+5+5410738377117:SRV' | Product identification of the goods using GTIN 5410738377117 |
| MEA+AAE+X7E+KGM:250' | Gross weight of returnable pallet plus 14 packages on the pallet is 250 Kilograms |
| PCI+33E' | Marked with the EAN.UCC serial shipping container code |
| GIN+BJ+354123450000000014' | Identification of marked serial shipping container code |
| UNT+23+ME000001' | Total number of segments in the message equals 23 |

*Illustration 1: On the left, an example of an EDI message (IFTMIN or instruction message). On the right, explanations for each line of the message example. [EAN, 2002]*

Currently the most popular ways to handle XML documents are to use languages such as XPath or XQuery. XPath's primary purpose is to address parts of an XML document. It also provides ways to manipulate strings, numbers and booleans. [Clark and DeRose, 1999] XQuery, like its name suggests, is an XML query language. Its purpose is to provide a way to retrieve and interpret information in XML documents. [Boag et al., 2007] Both of these language rely heavily on the user's knowledge of the structure of the document they are querying and as such are not the optimal tools for situations where the user does not know the content or structure of the documents.

All data is either unstructured, semistructured or structured. Most of the data we encounter is unstructured. For example, e-mails are unstructured data, because the body of the message is just freeform text. Unstructured data has no identifiable structure although a person can extract any relevant information from it. Images, videos and audio files are also unstructured data. Structured data, however, has identifiable structure. An example of structured data could be a typical database. The information in a database has to be organized based on its data model. It is also searchable by data type. Similar entities are grouped together and similar entities in the same group have the same descriptions. Lastly, semistructured data is data that has some sort of structure but that structure is not necessarily always the same. For structured data a schema level

definition can be made that the instance level data conform to, whereas semistructured data are irregular and incomplete and whose structure is frequently changing in an unpredictable way. Therefore is is typical of semistructured data that in it data belonging to the schema and instance levels are mixed.

Data in XML format fall into the semistructured category. The tagged format of XML documents gives the data some structure but the order of elements might not be the same or the elements might not have all the same attributes.

The purpose of this thesis is to present another way to look at XML documents. Because the documents are typically semistructured, there is a need to handle documents where the structure is unknown. In complex and large documents, which are previously unfamiliar to the user, it is a hard task for the user to find out manually (i.e. by reading textual documents) the content and structure of the document of interest.

This thesis introduces a Prolog-based prototype for analyzing XML documents where their contents and structures are not known beforehand. The user has a set of logic programming predicates to use, with which he can analyze the structure and content of any XML document without having to browse through the document.

The structure of the thesis is as follows. Chapter 2 discusses the basics of XML and DTD, XML schemas and RDF. Chapter 3 is about the problems related to the manipulation of XML. The fourth chapter describes the XML relation, which is the foundation on which the prototype in this thesis was created. Chapter 5 discusses how the XML relation and logic programming can be used together and in Chapter 6 the reasons for developing a tool for XML analysis are explained. In Chapter 7 an example document is given and the use of the analysis predicates is demonstrated. Chapter 8 features a comparison of traditional query languages with the prototype. Ideas for further development of the prototype are given in Chapter 9. Discussion of other work on XML representation and XML query languages is presented in Chapter 10 and Chapter 11 contains the conclusions of this thesis.

# 2. XML

## 2.1. Basics of XML

An XML document is composed of data items called attributes and elements. The data items are named and a specific data item may have several instances within an XML document.

An element consists of start and end tags, which declare the nature of the data between them. There are no predefined tags; they can all be defined by the user, thus making the language extensible. XML tags differ from, for example, HTML tags in that they describe the semantics of data – not their presentation style. In other words, XML is self-describing and this helps applications on the web understand XML documents made by other applications [Gou and Chirkova, 2007]. However, understanding the semantics of a document can also include analyzing its structure. Depending on its place in the document, the tag `<title>` can relate either to an album or to a single song.

For example, if an XML document contains information on a book, the start tag could be `<book>` and the end tag `</book>`. The start and end tags must be named exactly the same, except that a forward slash is added to the end tag. This includes case sensitivity, i.e. `<book>` is not the same as `<Book>`. The tags also cannot contain white spaces.

An element can contain text or other elements. Typically an XML document contains elements within other elements, i.e. nested elements. Nested elements are used to describe the hierarchical relationships between elements. For example,

```
<book>
     <chapter>Chapter1</chapter>
     <chapter>Chapter2</chapter>
</book>
```

describes a book that contains two chapters. More nested elements can be added to gain the desired detail:

```
<book>
     <title>Book of XML</title>
     <chapter>
          <title>Basics of XML</title>
     </chapter>
     <chapter>
          <title>Advanced XML</title>
          <pages>23</pages>
     </chapter>
```

```
</book>
```

An attribute is information in terms of which a property related to a specific element is expressed. Sometimes attributes can also be called name/value pairs, because they consist of a name and a value in quotes. [Keogh and Davidson, 2005] Attributes are placed inside an element's start tag and unlike elements, they cannot be nested. There is no limit to the amount of attributes that an element can have. However, each attribute within an element's start tag must have a unique name. Continuing with our previous example:

```
<book published="2011" totalPages="50">
     <title>Book of XML</title>
     <chapter startPage="1">
          <title>Basics of XML</title>
     </chapter>
     <chapter startPage="27">
          <title>Advanced XML</title>
          <pages>23</pages>
     </chapter>
</book>
```

The attributes that were added to the example add information to the document. The added information might as well have been described with elements, but some developers prefer using attributes for certain information either for the sake of readability or because the information concerns the whole element.

An XML document has a certain hierarchy. Each document must have one root element, i.e. an element that contains all other element occurrences in the document. In other words, the root element has only one instance in any XML document. The root is also the parent of all the other elements. Parent, child and sibling are terms that are used to describe the relationships of the elements. In our example, `<book>` is the root element. The elements within the `<book>` element are its children, of which there are three in this case: one `<title>` and two `<chapter>` element occurrences. On the other hand, the `<book>` element is the parent of these element occurrences. Both `<chapter>` element occurrences also have children. The first one has one while the second has two. Element occurrences that are immediately dependant on the same element occurrence are called siblings.

As the term root might imply, XML documents can also be seen as tree structures. When the tree is actually drawn, it is usually drawn upside down with the root element at the top. The child elements are visualized as branches. Elements without any child elements are visualized as leaves. The example used above is illustrated below (Illustration 2). Attributes are drawn the same way as elements, but they are marked

with a prefix @, for example `@totalpages`. The tree in Illustration 2 is based on the node-labeled model, which labels the nodes. Another model is the edge-labeled model, where the labels are associated with the edges [Gou and Chirkova, 2007]. In Illustration 2 an example of a branch element would be `title` and all the values, e.g. `'Book of XML'`, are leaf nodes.

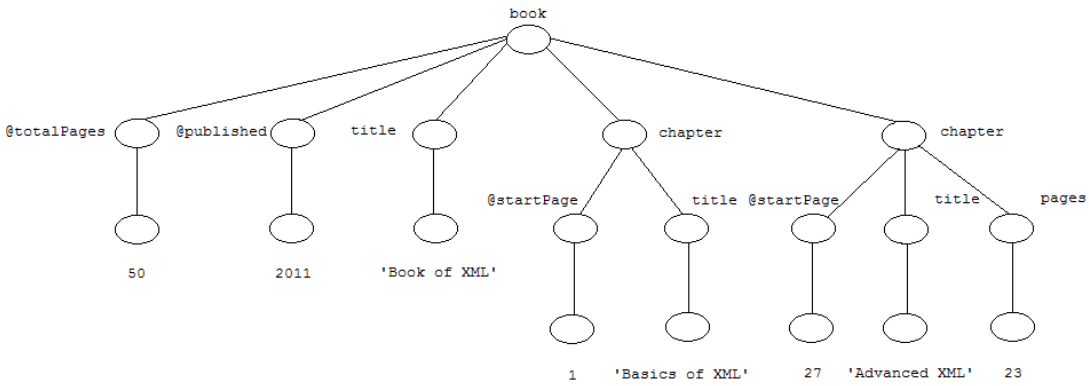


*Illustration 2: Data tree of the example XML document.*

One of the great advantages of XML is its flexibility. There is no need to set fixed length constraints on the data, because the start and end tags express where a specific piece of data starts and ends. In contrast, position based documents require strict constraints on the length of each data field, because the whole document will become unmanageable if the start point of a specific data is not where it should be.

The example above also illustrates another kind of flexibility that XML provides. It is semistructured as discussed earlier, so each instance of an element in an XML document can be organized differently. The example has two chapter elements. The first one has only one child element, while the second has two. This irregularity is very common in XML documents.

Because an XML document is plain text, it can be handled by any system capable of handling text. XML is a language made to represent information, not to query or manipulate it. The document itself does not do anything, but a program is needed for manipulating the data in the XML file. An XML file can be quite easily transformed to another form using XSLT (Extensible Stylesheet Language Transformations) [Clark, 1999]. This cannot be said of text documents in general. In XML the tags provide start and end points to the text between them. By finding tags `<name>` and `</name>` the exact locations with that name can be found and the data extracted. In a plain text document this would not be possible.

## 2.2. DTD, XML schema, RDF

Because of its simplicity and versability, XML has become very popular in transferring information between businesses. Because different companies have different ways and applications to handle their data, XML is useful, because it provides a format that can be read with any text processor. Unlike other markup languages, such as HTML, XML does not have a single standard that is used for all documents. Companies can create a standard of their own to use in business to business communication between them. This can be done by using a DTD (Document Type Definition) [Bosak et al., 1998].

A DTD is used to define the content and potential structural alternatives in a specific XML document or a collection of XML documents. It declares which element and attribute instances may appear in which mutual relationships in the document. It does not necessarily define the exact structure of the document, is just tells us what the structure may be. An example of a DTD is shown below (Illustration 3).

```
<!ELEMENT note (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

*Illustration 3: An example of a DTD [W3Schools.com].*

The companies doing business together agree on some rules for the structure of the XML documents they exchange as well as the names for different elements. These rules are represented based on the DTD. All partners can then use the DTD to verify the validity of the documents they receive from each other. If several businesses in the same field share information, the DTDs they use form a sort of vocabulary for this industry sector. Because the specific words or phrases have to be used in the XML documents to communicate with other businesses, they gradually become the terms used in other communication as well. However, A DTD does not in any way define the values in the element or attribute occurrences, it only names the data units and defines potential interrelationships among their instances.

An alternative for using DTDs is the XML schema [Fallside and Walmsley, 2004]. The schema is created with the XML schema language or XML schema definition (XSD). Everything that can be done with a DTD can also be done using an XML schema and the schema also provides more functionality. The DTD example that was used earlier

can be made into a schema as shown in Illustration 4. Whereas DTDs can be used to define the structure of the XML document, schemas can be used to also define the type of data in the document. For example, an XML schema can define that the data within the `<Birthday>` tags is of the type date. This way the element can only contain data that confirms to the definition of the date datatype.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns="http://www.w3schools.com"
elementFormDefault="qualified">

<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

*Illustration 4: An example of a schema [W3Schools.com].*

Yet another way to describe relationships between data units is RDF (Resource Description Framework) [Klyne and Carroll, 2004]. It is a standard framework for representing information in the Semantic Web. It uses triples that consist of a subject (the information resource to be described), a predicate (a property) and an object (the value of the property). A set of triples forms an RDF graph, which can be illustrated by a node and directed-arc diagram (Illustration 5) or in the standard RDF/XML format (Illustration 6). RDF is one of the most popular ways of describing data in the Semantic Web. It is used to define the relationships between different data units within an XML document. Unlike DTD and schemas, it is not used to validate the document, just to add more metadata and to help the user make sense of the document at hand.

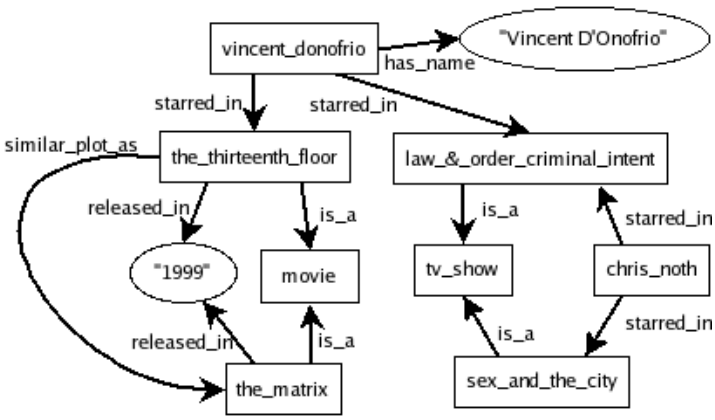


*Illustration 5: An example of an RDF graph [Tauberer, 2006].*

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:ex="http://www.example.org/">
    <rdf:Description rdf:about="http://www.example.org/vincent_donofrio">
        <ex:starred_in>
            <ex:tv_show rdf:about="http://www.example.org/law_and_order_ci" />
        </ex:starred_in>
    </rdf:Description>
    <rdf:Description rdf:about="http://www.example.org/the_thirteenth_floor">
        <ex:similar_plot_as rdf:resource="http://www.example.org/the_matrix" />
    </rdf:Description>
</rdf:RDF>
```

*Illustration 6: The RDF/XML representation of the RDF graph example [Tauberer, 2006].*

DTDs and XML schemas can be used to validate XML documents, in which case the document must confirm to the rules of the DTD or schema. However, the DTDs and schemas can be constructed so that they are more like instructions. The presence of either does not guarantee that the documents that use them will necessarily follow the rules.

## 3. Problems related to the manipulation of XML

There is a need to consider XML documents from several perspectives. At the moment there are two basic ways to handle XML documents: data-centric and document-centric. The data-centric approach is used to handle highly structured documents where the focus is on the data itself and not so much on the order of elements and attributes in the document. Document-centric approaches on the other hand are used to handle documents that are very unstructured and the order of elements is important [Lu et al., 2006]. Kamps et al. state in their paper [2006] that while query languages such as XPath and XQuery can be very effective for querying data-centric XML, another approach seems to be needed for querying document-centric XML.

At the moment XML languages such as XPath and XQuery require the user to be familiar with the structure and content of the document because of their path-oriented nature. To get the needed information out of the document, the user has to know the path to its exact location in the document. This is especially problematic with large XML sources, especially if some data are repeated on different hierarchical levels. Some files have a DTD to help with defining the structure but in many cases the DTD is too long and complex so that it is of no help. Some DTDs can also be incomplete and thus fail in their purpose.

According to Niemi and Järvelin [2006] the traditional way to handle unknown structures and contents is to use plain keyword search. The problem with this approach is the way keyword-based searches produce inexact results to complex information needs. Searching with the word ”master” in a document that contains data about the university can bring up results that contain information about master's degree studies as well as information about people with a master's degree that work at the university. In some cases the word ”master” might even occur in course titles. Erwig [2003] also states that search engines based on keywords are not powerful enough to exploit the structure that the XML format contributes to data.

The problem with path-oriented XML languages in general is that they are not suitable for ordinary end users. As Erwig states in his paper [2003], it cannot be expected that end users will be able or willing to learn sophisticated XML query languages. End users need a query language that is easy to use and does not require detailed knowledge of the document's structure. XML is used in so many business areas, that most of the end users that query XML documents for information are not necessarily very adept with

computers. This leads to them not being able to quickly look at a document and decipher the path to the needed knowledge. Also, the pattern matching mechanism related to path-oriented querying and manipulation is one of the biggest obstacles for the usual end user. It requires an understanding of procedural variables and the ability to use them, which are not skills that most end users possess.

Even if the document has a DTD that is not incomplete or too complex, the DTD does not give any information on the values used in the elements or attributes. Using acronyms such as DOB as element names can create problems in these cases. One person may quickly understand DOB to stand for ”date of birth” while another may have no idea what it means. In business to business communication this can become a problem also because communication might be conducted in English while the end users are not native English speakers. By looking at the values in the DOB elements, it might become easier for the user to interpret the semantics of the DOB elements. The values in the elements can reveal a lof of the semantics of the document that is not explicitly expressed in it. For example, if a document handling music has a data item named ”composer” and the values in the instances of that data item are names of classical composers, it implicitly reveals that the document handles classical music even if this is not explicitly mentioned in the document.

Another problem arises from the way information is stored. Relational databases are capable of storing and processing large volumes of data [Florescu et al., 2000]. Because of this relational databases are the common way to store data, while XML is the most common way to transfer data. Therefore there is often the need to convert the data from the XML documents to the relational database.

The data in a database has to be structured. The data in documents, however, is often not as structured. Niemi and Järvelin say in their paper [2006] that data in documents is often irregular, incomplete and with a changing structure. The transformation of data in a relational database to an XML-based document is relatively easy, whereas converting XML documents unambiguosly to data in a database is not easy at all. Usually the order of the elements is lost in the conversion from the XML document to the relational database and as a result, it is very hard to restore an XML document with its original structure from a relational database [Lu et al., 2006].

From the document-centric view, an easy way would be to store the whole document in one single data item. That would, however, mean that the document would always have to be handled as a whole. From the data-centric view, the document could be broken

into pieces which would then be stored as separate data items. This way the pieces could be manipulated but storing the document and also restoring it to its document form would require quite a lot of effort. [Fiebig et al., 2002]

The problem with restoring documents back to their original form can be solved by using the XML relation representation created by Niemi and Järvelin. Based on their notion of the relation it is possible to define the transformation process from textual XML data in a document to relationally organized losslessly, i.e. no information about the structure is lost. Every data item name instance and a single word in its value has a unique index in the XML representation. Due to this property, the XML representation of a document can easily be restored to its original textual form.

# 4. The XML relation

Niemi and Järvelin [2006] presented a novel way to represent XML data, because they felt that the traditional way of representing XML documents as directed labeled graphs led to numerous undesirable features. The traditional approach includes, among others, complex path-oriented XML query languages as well as the mismatch between XML data and relational databases. Niemi and Järvelin proposed that each XML document would be represented as an XML relation with the schema **D(C, T, I)**. In the schema **D** is the name of the document, **C** is a component in the document, **T** describes its type and **I** is its unique index.

Components include each element name occurrence, attribute name occurrence and values in element or attribute occurrences. If a value is a string consisting of multiple words, then each word is treated as a separate component. The type of a component can be element, attribute or value expressed by the letters 'e', 'a' and 'v', respectively. Each of the components has an unambiguous index, which indicates the location of the component within the document. This way, the relation can be converted back into the original textual representation because the structural information is retained in the indices. In fact, the path to a specific piece of information is still stored in the index, although the user cannot see it.

## 4.1. Constructor algebra

Niemi and Järvelin give the constructor algebra for the XML relations in their paper [2006]. The two main features of the algebra are that it yields a relational representation for any XML document and that the operations automatically re-index the indices to reflect the structure of the result document. Furthermore, the algebra has the closure property, i.e. each of its operations produces the XML relation which can be used as an operand in other operations.

Before the definition of the algebra, we need to address some basic notations used in it:

1. The length of a tuple *t* is denoted by *len(t)*. For example, *len*((a, b, c)) = 3.
2. The index *ind* is represented between angle brackets and it is divided into two parts as follows: *ind* = <*part1* ⊥ *part2*>. *Part1* refers to those elements which belong to the first part in *ind* whereas *part2* is the index consisting of the rest of the

components. In the context of indices a letter refers only to a single index component whereas the symbol ξ is used to refer to one or more index components. For example, if *ind* = <1, 3, 1, 4> then the expression <i ⊥ ξ> means that *i* refers to the first component in *ind*, i.e. i = 1 and ξ is the index <3, 1, 4>. Thus, the expression < ξ ⊥ j> applied to *ind* yields ξ = <1, 3, 1, 4> and j = 4.

*Definition 1*: An XML relation is constructed recursively by finite application of the following rules:

(1) Let *c* denote the value of an attribute or an element. If the value at hand consists of words, then *c* denotes a single word in it. In these cases *c* is represented as an XML relation *{(c, 'v', <1>)}*. In the tuple *'v'* indicates that *c* is a value or belongs to some value. In the latter case c is a word in some value.

(2) An attribute name *an* is represented as an XML relation *{(an, 'a', <1>)}*. In the tuple *'a'* indicates that *an* is an attribute name.

(3) An element name *en* is presented as an XML relation *{(en, 'e', <1>)}*. In the tuple *'e'* expresses that *en* is an element name.

(4) If $R_1$ and $R_2$ are two XML relations, then the concatenation constructor $R_1 <> R_2$ constructs an XML relation

$$R_1 \cup index\_transformation(maxfirst(R_1), R_2)$$

where

$$maxfirst(R) = |\{(c, t, ind) \mid (c, t, ind) \in R: len(ind)=1\}|$$

i.e. *maxfirst(R)* expresses the number of those indices in *R* whose length is 1.

$$index\_transformation\ (int, R) = \{ (c, t, <i + int \perp \xi>) \mid (c, t, <i \perp \xi>) \in R\}$$

i.e., the function *index_transformation* re-indexes the tuples in *R* by summing integer *int* with the first element of an index. In the above formula *t* denotes any component type, i.e. t ∈ {'a', 'e', 'v'}.

(5) If $A$ represents an attribute name as an XML relation (see rule (2) above) and $R$ its content as an XML relation, then the attribute constructor denoted by $A \ominus R$ constructs an XML relation $A \cup \{(c, t, <1 \perp ind>) \mid (c, t, ind) \in R\}$. In other words the length of each index in R is added by one by inserting '1' as the first component in the indices.

(6) If $E$ represents an element name as an XML relation (see the rule (3) above) and $R$ its content with possibly (nested) substructure as an XML relation, then the element constructor denoted by $E \omega R$ constructs an XML relation $E \cup \{(c, t, <1 \perp ind>) \mid (c, t, ind) \in R\}$.

### 4.2. Example

To illustrate how an XML document is transformed into the corresponding XML relation representation, we use the following very small XML document (called Sample).

```
<dvds>
     <dvd discs="2" run_time="177">
          <name> Kamelot - One Cold Winter's Night </name>
          <genre> Music </genre>
     </dvd>
</dvds>
```

Based on the first three rules of the algebra, in Table 1 we give the basic information for structuring the XML relation. We also give each component a notational abbreviation to help with the rest of the demonstration.

| **Element names** | **Attribute names** | **Values** |
|---|---|---|
| E1 = {('dvds', 'e', <1>)} | A1 = {('discs', 'a', <1>)} | V1 = {('2', 'v', '<1>')} |
| E2 = {('dvd', 'e', <1>)} | A2 = {('run_time', 'a', <1>)} | V2 = {('177', 'v', <1>)} |
| E3 = {('name', 'e', <1>)} | | V3 = {('Kamelot', 'v', <1>)} |
| E4 = {('genre', 'e', <1>)} | | V4 = {('-', 'v', <1>)} |
| | | V5 = {('One', 'v', <1>)} |
| | | V6 = {('Cold', 'v', <1>)} |
| | | V7 = {('Winter's', 'v', <1>)} |
| | | V8 = {('Night', 'v', <1>)} |
| | | V9 = {('Music', 'v', <1>)} |

*Table 1: Basic components related to the example document, Sample.*

In terms of the constructor algebra we can produce the XML relation representation corresponding to the example document by the following sequence:

```
E1 ω (
     E2 ω (
          (A1 ϴ V1) <>
          (A2 ϴ V2) <>
          E3 ω (
               V3 <> V4 <> V5 <> V6 <> V7 <> V8
          ) <>
          E4 ω V9
     )
)
```

Now we consider its construction by starting from the innermost part of the sequence,

```
V3 <> V4 <> V5 <> V6 <> V7 <> V8 (denoted by I).
```

By applying rule 4 for evaluating `V3 <> V4` means that the expression

```
V3 ∪ index_transformation(maxfirst(V3), V4)
```

has to be performed. In it `maxfirst(V3)` returns 1 as the result. Based on this value the function

```
index_transformation(1, V4)
```

yields the set

```
{('-', 'v', <2>)}.
```

After the evaluation of all the construction operations in I, we get the XML relation described in Table 2. As can be seen from Table 2, the indices have re-indexed to express the order of single words in the value, or in this case the order of the words in the DVD name.

| |
|---|
| {('Kamelot', 'v', <1>), |
| ('-', 'v', <2>), |
| ('One', 'v', <3>), |
| ('Cold', 'v', <4>), |
| ('Winter's', 'v', <5>), |
| ('Night', 'v', <6>)} |

*Table 2: The XML relation (denoted by II) yielded by the operation sequence I.*

The next part in the sequence is

```
E3 ω (II) <> E4 ω V9 (denoted by III).
```

In the evaluation of this sequence the sixth rule of the algebra is also needed. The evaluation gives the XML relation in Table 3.

| | |
|---|---|
| {('name', 'e', <1>), | ('Kamelot', 'v', <1, 1>), |
| ('genre', 'e', <2>), | ('-', 'v', <1, 2>), |
| | ('One', 'v', <1, 3>), |
| | ('Cold', 'v', <1, 4>), |
| | ('Winter's', 'v', <1, 5>), |
| | ('Night', 'v', <1, 6>), |
| | ('Music', 'v', <2, 1>)} |

*Table 3: The XML relation achieved by the operation sequence III.*

The constructor expression

```
(A2 ϴ V2)
```

yields the XML relation

```
{('run_time', 'a', <1>), ('177', 'v', <1, 1>)}.
```

The rest of the operations in our original sequence are evaluated analogously and it produces the XML relation representation for our sample document Sample in Table 4.

| |
|---|
| Sample('dvds', 'e', <1>) |
| Sample('dvd', 'e', <1, 1>) |
| Sample('discs', 'a', <1, 1, 1>) |
| Sample('2', 'v', '<1, 1, 1, 1>') |
| Sample('run_time', 'a', <1, 1, 2>) |
| Sample('177', 'v', <1, 1, 2, 1>) |
| Sample('name', 'e', <1, 1, 3>) |
| Sample('Kamelot', 'v', <1, 1, 3, 1>) |
| Sample('-', 'v', <1, 1, 3, 2>) |
| Sample('One', 'v', <1, 1, 3, 3>) |
| Sample('Cold', 'v', <1, 1, 3, 4>) |
| Sample('Winter's', 'v', <1, 1, 3, 5>) |
| Sample('Night', 'v', <1, 1, 3, 6>) |
| Sample('genre', 'e', <1, 1, 4>) |
| Sample('Music', 'v', <1, 1, 4, 1>) |

*Table 4: The XML relation representation of the example document.*

# 5. XML relation in logic programming

Prolog [Colmerauer, 1990] is the main logic programming language. A program is a set of axioms or rules, which define the relations between objects. The program is used by running a query over the relations. If the query is found to be true, it is a logical consequence of the program.

Prolog has only one data type, which is called the *term*. Terms can be *constants*, *variables* or *compound terms*. A constant is an atom or a number. Atoms begin with a lower-case letter or they are in single quotes. Some examples of constants are 2, x and 'Kirk'. Variables always begin with an upper-case letter, which is why atoms beginning with an upper-case letter have to be separated from variables by single quotes. Variables are any objects, that have not been explicitly expressed, in the closed world represented in a Prolog program. A compound term is composed of a functor name and a number of arguments. The name of the functor has to be an atom and the arguments are terms. Complex objects in Prolog are represented by nesting terms of different types.

In a logical sense, a Prolog program consists of Horn clauses. There are three types of clauses: rules, facts and queries. A rule is of the form

```
Head :- Body.
```

This means that *Head* is true if *Body* is true. The *Body* consists of goals, which are calls to predicates. An example of a rule is

```
father(X, Y) :- parent(X, Y), male(X).
```

The above rule could be interpreted as follows: “X is Y's father if X is Y's parent and X is male”.

A fact is a clause without a body. An example of a fact is

```
male(john).
```

In it the Prolog programmer wants to express explicitly that John (an object) is male (i.e. a property related to John).

In this research an XML document is converted first into an XML relation by applying the constructor algebra discussed earlier and after that the data are stored in a PostgreSQL database. One might argue that because the data are now in a relational database, SQL (Structured Query Language) [Chamberlin and Boyce, 1974] can be used to gather information rather than creating a whole other way of doing it. However, SQL was tailored to extract and select data from previously familiar relations – not to analyze previously unfamiliar relations. In addition, XML data are fundamentally different than relational data and therefore SQL is not appropriate for XML [Deutsch et al., 1999].

The main difference between data in XML and data in a relational database is that XML is not rigidly structured. In a relational database, every data instance has a schema, which describes its content and structure. The instance level has been organized according to this schema. In other words, data are represented at two abstraction levels. In XML, however, the schema is represented with element and attribute names, which are mixed with their values. This makes XML data self-describing and it can model irregularities unlike the relational model. Thus a query language that is designed for relational, structured data is not as useful with semistructured XML data as a language or tool created for use with XML, although it can be used to some extent.

To use the XML relation with logic programming, it has to be presented in a format that can be handled with Prolog. First the data must be retrieved from the database. Prolog itself cannot be used to access the database and retrieve the data, so a short Java [Gosling et al., 2005] program had to be written to achieve this. Prolog and Java can be used together by using a library called JPL [Singleton et al., 2004]. JPL is a bidirectional Java/Prolog interface. Using JPL enables the user to embed Prolog in Java code or Java in Prolog code. The latter was used here so that the Java program could be called from the Prolog main program.

The following is the code for the retrieval of the data from the database by calling the Java program from the Prolog main program. The first `get_facts` predicate is used, when everything goes well and the connection to the database is achieved. The second predicate is there only to notify the user, if the connection fails and the information cannot be retrieved.

```
get_facts(Table) :- clear_all_facts, jpl_new('DBAccess', [],
DBA), jpl_call(DBA, getConn, [], ConnOk),
jpl_is_true(ConnOk), jpl_call(DBA, getAllData, [Table],
AllData), jpl_call(DBA, closeConn, [], _),
jpl_array_to_list(AllData, ADList), retract_in_use,
assertz(in_use(Table)), process(ADList), !.
```

```
get_facts(_) :- clear_all_facts, jpl_new('DBAccess', [],
DBA), jpl_call(DBA, getConn, [], ConnOk),
jpl_is_false(ConnOk), write('Could not get connection.'), nl.
```

To begin analyzing the document, the user calls the Prolog predicate `get_facts(Table)`. Because the prototype can only handle one document at a time, first all facts from a possible previous document are cleared (using the predicate `clear_all_facts`). To use the Java code from Prolog, an instance of the Java class has to be created from Prolog. For this, the system predicate `jpl_new` is used with the classname (in this case DBAccess), any parameters for the constructor (none here, which is why there is an empty list) and a variable that will be bound to the new reference (DBA here). Next, the predicate `jpl_call` is used to call the `getConn` method, this time using the variable created in the previous step. The `getConn` method is used to open to connection to the database. Again there are no parameters to pass to the Java method, but this time the method returns a truth value through the variable `ConnOk`. If the truth value is true, the program moves on to the next goal. However, if `ConnOk` is not true, the program moves to the second `get_facts` rule and notifies the user of the connection failure.

If the connection is open, then the method `getAllData` actually retrieves the data. The parameter `Table` is the name of the document (and thus, the name of the table in the database) to be analyzed. The Java method then simply gets the data from the given table. After that the data is converted from the query result into a form that is easier to handle with Prolog and then returned to the main Prolog program.

The data is returned by initializing the *`AllData`* variable as an array. The connection to the database is then closed and the array is converted into a Prolog list. The predicate `retract_in_use` removes any fact in the form of `in_use(table)` and `assertz` adds the new `in_use` fact to memory. This fact is used to store the name of the document at hand. Finally the data is transformed into a collection of facts with the following structure:

```
table_name(component, type, index).
```

The facts are stored in memory. Each fact contains the information related to one row (tuple) of the XML relation. In other words, the number of facts in memory is the same as the number of rows in the XML relation. This collection of facts is used to analyze the underlying XML document.

To illustrate, let's use the same example that was used in the previous chapter. The name of the small XML document is Sample, so the call to retrieve that document from the database would be the following:

```
get_facts('Sample').
```

The resulting data collection that would be stored in memory and not shown to the user is the same as in Table 4:

```
sample('dvds', 'e', <1>)
sample('dvd', 'e', <1, 1>)
sample('discs', 'a', <1, 1, 1>)
sample('2', 'v', '<1, 1, 1, 1>')
sample('run_time', 'a', <1, 1, 2>)
sample('177', 'v', <1, 1, 2, 1>)
sample('name', 'e', <1, 1, 3>)
sample('Kamelot', 'v', <1, 1, 3, 1>)
sample('-', 'v', <1, 1, 3, 2>)
sample('One', 'v', <1, 1, 3, 3>)
sample('Cold', 'v', <1, 1, 3, 4>)
sample('Winter's', 'v', <1, 1, 3, 5>)
sample('Night', 'v', <1, 1, 3, 6>)
sample('genre', 'e', <1, 1, 4>)
sample('Music', 'v', <1, 1, 4, 1>).
```

## 6. Reasons for analysis

As XML is increasingly popular as a means of data transfer, more and more people with no particular knowledge of XML find themselves in situations where they are required to handle XML documents. They do not master XML query languages such as XPath or XQuery but are required to retrieve data from some available unfamiliar documents, that can be quite large and complicated. In addition, even users familiar with XML query languages need to have some knowledge of the document structure to be able to use the languages to query previously unfamiliar documents.

There is a need for a tool in terms of which it is possible to analyze the structures and contents of XML documents that have no definition, such as a DTD or schema, or have a definition of some sort that is too complicated. In addition, documents can be so large that comprehending the purpose of the document requires a tool to help understand the structure and semantics of the document. For this reason in this thesis a set of Prolog predicates is developed to analyze XML documents based on their XML relation representations.

The predicates can be divided into three categories:

1. **Analysis of structure**. These predicates are used to analyze the structure of the document. They give information about the elements and attributes in the document, e.g. the amount of occurrences of an element with a specific name in the document or a listing of all the different attribute names.

2. **Analysis of content**. These predicates provide information about the values, i.e. the actual data of the document. They can be used for example to determine the maximum value of an attribute or if all the values in a certain element or attribute are presented uniformly. In addition, they can be used to help understand the semantics of the document. These predicates would be used to examine the values to check whether or not the document contains information that the user is interested in.

3. **Aggregate**. The structure analysis predicates can be used together with the content analysis predicates to gather information that is not directly given in the document. For example, percentages and average values for numerical values of elements or attributes can be calculated this way.

If the user does not know the structure of the document beforehand, its utilization is difficult, and often impossible. With the traditional, path-oriented query languages the user has to be familiar with the content and structure of the documents they are handling. This means that they would have to read the defining DTD or schema, which in turn means that they would have to be able to understand them. If no DTD is available, the user would need to browse the document itself, which is a major task if the XML document is a large one or if the user is handling a collection of documents. Thus, even if the user would be familiar with XPath or XQuery, without an accurate understanding of how the data is structured in the document, the user would be unable to issue meaningful queries [Barg and Wong, 2003]. Even with the use of wildcards, i.e. characters used to replace any element (usually an asterisk (*)), the user has to know the name of the destination element. In addition, when using, for example, XPath, the user has to know if the information they are looking for is in an element or an attribute. This is one example of a situation where even someone familiar with XPath would find the analysis predicates above useful.

With the predicates characterized above, the user can find out what elements, attributes and values the XML document consists of and how they relate to each other without having to go through the document manually. Using this information they can then further analyze the underlying document. If a certain element or attribute name seems ambiguous or the name is an unfamiliar acronym, the user can look at the values in that element or attribute to get more intuition on the semantics related to the attribute or element name. With the data they can gather of the document using the predicates they can then move on to use XPath or XQuery with more ease, if needed.

All of this relates closely to the concept of dataspaces [Franklin et al., 2005]. In their paper introducing the idea of dataspaces, Franklin and his colleagues outline a need to have a new abstraction for data management. The biggest challenge of information management today is that organizations rely on a large number of diverse, interrelated data sources but have no means of managing their dataspaces. For this purpose the idea of DataSpace Support Platforms (DSSP) is suggested. Instead of data integration, dataspaces shift the emphasis to data co-existence. The goal of DSSP is to provide base functionality over all data sources, regardless of how integrated they are. In other words, although the information is stored in different ways in different data manegement systems, there should be a way to query or search all of the data in one go. The results are perhaps not very accurate, but by gradually refining the query the user can locate the desired answers.

The way dataspaces relate to the work in this thesis is that the data in XML documents can be presented in so many ways that a tool that is able to analyze the documents structure and content is needed. By using this tool the user's knowledge of the data grows up to the point where usage of other tools is possible. As more and more data is stored in XML format or can easily be transformed to XML, the need only grows. The prototype described here is one example of a tool that could be used to satisfy that need. It could be said that the prototype can be used like database profiling for XML documents, i.e. ”XML profiling”. Database profiling can be described as ”analysis of the structures and properties exposed by an information source” which allows for assessment of the utility and importance of the database as well as determining the structure of the database in preparation for specific data applications [Howe et al., 2008]. This is exactly what the prototype is used for.

# 7. Example

To illustrate the expressive power of the analysis predicates, a short example document is introduced next.

The example document contains information about DVDs. There are seven DVDs in the document and each one has some detailing information listed. Each DVD has either a title or a name. In case of movies, the other information might include the names of the directors, writers or actors. Also a tagline or genre might be given. Some numeral information include the number of discs, year of release and running time in minutes.

## 7.1. Example document

```
<dvds>
     <dvd discs=”1” year=”2005” run_time=”73”>
          <title> Corpse Bride </title>
          <director> Tim Burton </director>
          <actors>
               <actor> Johnny Depp </actor>
               <actor> Helena Bonham Carter </actor>
          </actors>
     </dvd>
     <dvd discs=”2” year=”1994” genre=”Thriller”>
          <title> The Stand </title>
          <writers>
               <writer> Stephen King </writer>
          </writers>
          <actors>
               <actor> Gary Sinise </actor>
               <actor> Molly Ringwald </actor>
          </actors>
          <tag_line> the end of the world is just the beginning
          </tag_line>
     </dvd>
     <dvd discs=”1” year=”1994” run_time=”137”>
          <title> The Shawshank Redemption </title>
          <director> Frank Darabont </director>
          <genre> Drama </genre>
          <writers>
               <writer> Frank Darabont </writer>
               <writer> Stephen King </writer>
          </writers>
          <actors>
               <actor> Tim Robbins </actor>
               <actor> Morgan Freeman </actor>
          </actors>
     </dvd>
     <dvd discs=”2” run_time=”177”>
          <name> Kamelot - One Cold Winter's Night </name>
          <genre> Music </genre>
     </dvd>
     <dvd discs=”2” run_time=”240”>
          <name> Iron Maiden – Live After Death </name>
```

```
        <genre> Music </genre>
    </dvd>
    <dvd discs="1" year="1998" run_time="115" genre="Thriller">
        <title> Blade The Daywalker </title>
        <director> Stephen Norrington </director>
        <writers>
            <writer> David S. Goyer </writer>
        </writers>
        <actors>
            <actor> Wesley Snipes </actor>
            <actor> Stephen Dorff </actor>
        </actors>
        <tag_line> It takes one to kill one </tag_line>
    </dvd>
    <dvd discs="1" year="2008" run_time="116" genre="Drama">
        <title> Gran Torino </title>
        <director> Clint Eastwood </director>
        <actors>
            <actor> Clint Eastwood </actor>
            <actor> Bee Vang </actor>
        </actors>
    </dvd>
</dvds>
```

The collection of facts that is generated from the XML relation corresponding to this example document can be found in Appendix 1. Let's assume that the name of the document is `info`.

The predicates created for the prototype in this thesis can be divided into different groups based on what they are used to analyze.

### 7.2. Predicates for structure analysis

If the user does not know anything about the structure, or even content, of the document of interest, the structure analysis predicates are often a good starting point to get more information on the underlying document.

With the structure analysis predicates the user gets information about the relationships among the different components of an XML document. Because XML is a self-describing language, the names of elements and attributes also shed light on the subject of the document.

Let us assume that the example document above is totally unknown to the user. S/he does not know anything of its structure or content. The first thing that might reveal the nature of the document is the name of the root element of the document. The predicate used to find this out is called simply `root` and its usage is similarly simple:

```
Query 1:
root.
```

The processing of this goal produces the result

```
dvds
```

in the context of the sample document. From this we can deduce that the document concerns DVDs. Of course there can be cases where the root element is named simply root, which does not say anything about the document. In a case like this the names of the elements on the second level of the hierarchy could be more useful. For this purpose the predicate `elements_level(Document, Level)` has been developed. It returns the names of all elements on the given level in the given document. Now, the user would like to know the names of all element occurrences under the root element, which is the second level. As our document is called `info`, the query becomes:

```
Query 2:
elements_level(info, 2).
```

The result for this query is

```
[dvd, dvd, dvd, dvd, dvd, dvd, dvd].
```

From the result of the previous query at least the user now knows that the document concerns DVDs. By looking at the result more closely, the user can see that the document at hand contains information about seven DVDs.

Now, a more comprehensive look at the document would tell the user more. He or she can use `show_all_elements` to find out what elements and attributes the document consists of. Like its name suggests, the predicate `show_all_elements(Document)` returns all element names found in the document.

```
Query 3:
show_all_elements(info).
```

In the context of our sample document it prints the following result:

```
actor
actors
director
dvd
dvds
genre
name
tag_line
```

```
title
writer
writers.
```

If the user wants to find out all attribute names in the document, he or she can use the predicate `show_all_attributes(Document)` for this purpose.

```
Query 4:
show_all_attributes(info).
```

Based on our sample document it produces the following printing:

```
discs
genre
run_time
year.
```

The user can utilize the information gained with the structural analysis predicates in analysing the content of the XML document.

### 7.3. Predicates for content analysis

The predicates described in this chapter are useful for content analysis. They are meant to be used to find out what kind of values are in the different elements and attributes in the document. Along with structure analysis, content analysis is another good starting point. From the content the user can see if the document handles the kind of information they are interested in.

For example, the user might be looking through several documents that contain information about music. Just by looking at element names, such as `title`, `composer` and `genre`, the user would not be able to tell what kind of music the documents concern. By looking at what the values are for the different elements, the semantics of the document would become clear. Also, because the same element name can be used for data of different kinds, checking the values is helpful. For example, `title` can be used as the element name for the title of a CD as well as the title for a single song.

One of the basic needs of a user is to know what values a certain element or attribute has. If the user has applied the above structure analysis predicates, he has some idea of the content of the document, but the actual values in the elements and attributes give a much clearer idea of it all. It is also possible that the element name is an acronym that the user does not recognize. Looking at the values the meaning could become clear. For example, DOB could be an element name. Just the name is a bit obscure but by looking

at the values, which would be dates, it could come apparent to the user that DOB stands for “date of birth”.

In the context of our example document, there might be questions concerning semantics related to some of the element or attribute names. For example, what exactly is meant by the attribute name `discs`? To find this out, the predicate `values(Document, DataItemName, Result)` is available. As a parameter the predicate takes the name of the document to perform the query on as `Document` and the name of the wanted element or attribute as `DataItemName`. As `Result` the predicate returns a list of the values which appear in different element/attribute occurrences in the document:

```
Query 5:
values(info, discs, Result).
Result = [1, 2, 1, 2, 2, 1, 1].
```

The query returns numbers so the attribute `discs` probably means the amount of discs the content is spread out to.

Another version of the values predicate can be used to look at certain elements/attributes that belong to a specific element. For example, the user can find out what genres are present among the DVDs by using the predicate `values(Document, ParentElementName, ChildDataItemName, Result)`. `ParentElementName` is the name of the element under which the `ChildDataItemName` should be found. `Result` is a list of the values found in those data items.

```
Query 6:
values(info, dvd, genre, Result).
```

In the context of our sample document, the above query returns the following:

```
Result = ['Thriller', 'Drama', 'Music', 'Music', 'Thriller',
'Drama'].
```

This predicate can be very useful when the user wants to concentrate on a specific data item and what kind of values can be found in its occurrences. Also, an attribute with the same meaning could be named differently in different element occurrences. This DVD list might consist of elements named `movie` and `live_recording`. In a case like this, if the user was only interested in the movies, s/he could use the above predicate to check the values of `genre` that occur under the instances of `movie`.

Let us assume the user is interested in DVDs with the genre “Drama”. With the help of the structure analysis predicates they can look at the list of element names and see that there seem to be 2 different elements that might contain the name of the DVD: `title` and `name`. The user wants to check both to see what the drama DVDs are called. For this purpose there is the predicate `get_data(Document, ParentDataItemName, ChildDataItemName, ChildDataItemValue, ResultDataItems, Result).` The predicate takes five parameters: name of the document at hand (`Document`), name of the parent element/attribute (`ParentDataItemName`), name of the element or attribute to search for under the parent data item (`ChildDataItemName`), the desired value for the named element/attribute (`ChildDataItemValue`) and a list of the data item names that the user wants to see the values for (`ResultDataItems`). `Result` is a list of the values for the data items in `ResultDataItems`. In this case the user wants to see the values of `name` and `title` from DVDs that have “Drama” as their genre in our example document `info`:

```
Query 7:
get_data(info, dvd, genre, 'Drama', [name, title], Result).
```

After processing this goal (question) the following is returned:

```
Result = [['Gran', 'Torino'], ['The', 'Shawshank',
'Redemption']].
```

All of the different DVDs might not contain the same data. This assumption is supported by the fact that the headings of the DVDs seem to sometimes be in the element `title` and other times in the element `name`. The user might want to find out if there is some data item that appears in all instances of the given element. The predicate `common_data(Document, DataItemName)` has been developed for exactly this purpose. The parameter `DataItemName` expresses the name of the targeted element.

```
Query 8:
common_data(info, dvd).
```

Based on the example document, the query returns the printing

```
discs.
```

Here `discs` was the only data item found in all instances of `dvd`. Another related predicate is `get_without(Document, ElementName, DataItemName, ResultDataItems, Result).` It is used to find the instance of `ElementName` where a

specific data item name `DataItemName` does not appear. For example, is there a DVD for which the genre has not been listed? We can make the following query to find out:

```
Query 9:
get_without(info, dvd, genre, [name, title], Result).
Result = [['Corpse', 'Bride']].
```

In our sample document the names of DVDs have been expressed in the data items name and title. After processing the query, `Result` expresses that the only DVD with no genre is titled “Corpse Bride”.

As it is now apparent that the data related to each DVD is somewhat varied, it might be interesting to know which DVD has the most data. In other words, the user can check which element occurrence has the most data items attached to it. For this purpose the predicate `max_data(Document, ElementName, ResultDataItems, Result)` can be used. It returns the values in the data items listed in `ResultDataItems` for the occurrence of the wanted element (`ElementName`) with the most data attached to it.

```
Query 10:
max_data(info, dvd, [name, title], Result)
```

gives the solution

```
Result = [[Blade, The, Daywalker]].
```

### 7.4. Predicates for aggregation analysis

Lastly our approach contains the predicates that can be used to gain data that is not presented explicitly in the document as an element, attribute or value. For example, the user can realize that the document consists of `dvd` elements. But how many are there? With small documents the user might just count the elements manually, but if we think of a typical DVD collection with tens or hundreds of DVDs, the task would be too time-consuming to be done by hand. In our approach c`ount(Document, DataItemName, Result)` is a predicate that counts the instances of the given `DataItemName`:

```
Query 11:
count(info, dvd, Result).
Result = 7.
```

It seems that there are seven instances of `dvd` in our sample document, `info`. Because the document is so small, this can be verified by actually counting the `dvd` elements manually from our textual sample document.

Other basic information the user might be interested in could include the maximum and minimum values among instances of such a data item whose instances have numeric values. In terms of the above content analysis predicates the user can find some such elements/attributes. For example, in our sample document `run_time` and `year` are such kinds of attributes. The predicates `max(Document, DataItemName, Result)` and `min(Document, DataItemName, Result)` are quite self-explanatory, i.e. they return the maximum and minimum values in the given `DataItemName`. The maximum run time, which appears in our sample document, info, can be found out by

```
Query 12:
max(info, run_time, Result),
```

which yields

```
Result = 240.
```

Correspondingly, the minimum run time can be found with the following query:

```
Query 13:
min(info, year, Result),
```

which returns

```
Result = 1984.
```

Based on the minimum and maximum predicates it is possible to find the DVD with the maximum or minimum value of some element or attribute. For example, now we know that the maximum run time among the DVDs is 240 minutes. If the user wants to know the name of the DVD with the maximum run time, then the predicate `get_max(Document, ElementName, DataItemName, ResultDataItems, Result)` can be used. The predicate returns the values of data items (`ResultDataItems`) belonging to the element (`ElementName`) with the maximum value of some element or attribute (`DataItemName`). The result can be something else than the name of the DVD, but in this case it seems most appropriate:

```
Query 14:
get_max(info, dvd, run_time, [name, title], Result).
```

The processing of the above query returns the following:

```
Result = [['Iron', 'Maiden', -, 'Live', 'After', 'Death']].
```

In addition to the minimum and maximum values, it is interesting to find out the average among the numeric values. For this purpose we define the predicate `average(Document, DataItemName, Result)`, which gives the average (`Result`) of the values related to `DataItemName` in the document `Document`. In the context of our sample document, the goal

```
Query 15:
average(info, run_time, Result)
```

produces the solution

```
Result = 143.
```

The predicate `higher_than_average(Document, ElementName, DataItemName, ResultDataItems, Result)` extracts from the underlying document the values of data items expressed in `ResultDataItems`, which can be found from element `ElementName` instances where the value of data item `DataItemName` is greater than the average of all values of this data item. If the user wants to find those DVDs whose run times are greater than average, s/he can do this by the following query:

```
Query 16:
higher_than_average(info, dvd, run_time, [name, title],
Result).
```

Its processing gives the solution

```
Result = [['Kamelot', -, 'One', 'Cold', 'Winter\'s',
'Night'], ['Iron', 'Maiden', -, 'Live', 'After', 'Death']].
```

These aggregate predicates as well as some of the other previously described predicates can also be used together to form more complex queries. These require the user to be familiar with variables. By using the same variable in different goals of a query, the user can combine different predicates with each other. As an example, the `max_data` predicate could be replaced by using two other predicates, `max_info(Document, DataItemName, Index)` and `show(Document, Index, ResultDataItems, Result)`, together. The predicate `max_info` gives the index for the `DataItemName` with the most data attached to it. This index is shared by the variable `Index`, which is a parameter for

the `show` predicate. It in turn finds the values of `ResultDataItems` for that DVD instance.

```
Query 17:
max_info(info, dvd, Index), show(info, Index, [name, title],
Result).
```

Now, because there are in fact two goals in the above query, we also get two solutions:

```
Index = [[1, 6]],
Result = [['Blade', 'The', 'Daywalker']].
```

The first result, `Index`, is the index of the DVD instance with the most data attached to it. The second result, `Result`, is the result for the whole query, i.e. the name of the DVD. When we compare this to the result of Query 10, we can see that they produced the same result.

As an example of how an even more complex query can be made, let us find out the percentage of those DVDs in the document that have a tag with the name `tag_line`. For this we can use a combination of the predicates developed in this thesis as well as basic arithmetic predicates available in a typical Prolog environment. It should be noted that this is a query that requires the user to be familiar with Prolog, but it shows how these predicates can be used by a user more skilled in Prolog.

```
Query 18:
count(info, dvd, Total), count(info, tag_line, Tags), Res1 is
Tags/Total, Percentage is Res1*100.
```

Based on our sample document, we get the following results:

```
Total = 7,
Tags = 2,
Res1 = 0.285714,
Percentage = 28.5714.
```

First we get the number of DVDs into the variable `Total`. Next we need the amount of tag lines in the document. This info goes to the variable `Tags`. After that we divide the amount of tag lines with the amount of DVDs. The result is expressed by the variable `Res1`. Finally the instantiated value `Res1` is multiplied by a hundred to get a percentage value for the user. This whole process illustrates that the predicates can be very flexible tools in the hands of someone who is more familiar with Prolog.

# 8. Comparison with traditional query languages

To illustrate the differences between traditional query languages and the prototype developed in this thesis, some sample information needs are specified with both approaches.

For the comparison, the following three queries are considered:

1. Give all writers expressed under the element writers.
2. Give all values in the element director instances.
3. Give all DVDs with 2 discs.

For the purpose of these comparisons, assume that the current context is the root element of the `info` document introduced in the previous chapter and that the document has been saved in the file info.xml.

## 8.1. XPath and XQuery syntaxes

XPath is a query language that uses paths to select nodes (elements/attributes) from XML documents. An XPath query is a sequence of alternating axes and tags. Two most commonly used axes are the child axis / and the descendant axis //. An example of using the child axis is `A/B` where child nodes `B` of parent nodes `A` are selected. `A//B` on the other hand denotes selecting `B` descendant nodes of `A` nodes, i.e. all `B` nodes anywhere under `A`. [Gou and Chirkova, 2007]

An absolute path is a path that points to the same location in the document, no matter what the current context is. A relative path, however, is relative to the current location. In XML documents the same element name can appear in different locations, so a relative path can be used to locate that info without explicitly stating the path to it.

XQuery is more expressive than XPath. An XQuery query consists of For-Let-Where-Return (FLWR) clauses. The For and Let clauses use XPath expressions to bind nodes to user-defined variables. The Where clauses specify the selection or join predicates on the variables. The Return clauses operate on variables to format query results in the XML format. [Gou and Chirkova, 2007]

## 8.2. Comparisons

### 1. Give all writers expressed under the element writers.

**XPath:**

Absolute path:

```
/dvds/dvd/writers/writer
```

Relative path:

```
descendant::writers/child::writer
```

**XQuery:**

```
for $x in doc("info.xml")/dvds/dvd/writers/writer
     return {data($x)}
```

**In our approach:**

```
values(info, writers, writer, Result)
```

### 2. Give all values in the element director instances.

**XPath:**

Absolute path:

```
/dvds/dvd/director
```

Relative path:

```
descendant::director
```

**XQuery:**

```
for $x in doc("info.xml")/dvds/dvd/director
     return {data($x)}
```

**In our approach:**

```
values(info, director, Result)
```

### 3. Give all DVDs with 2 discs.

**XPath:**

Absolute path:

```
/dvds/dvd[@discs = 2]/title or /dvds/dvd[@discs = 2]/name
```

Relative path:

```
descendant::dvd[attribute::discs = 2]/child::title or
descendant::dvd[attribute::discs = 2]/child::name
```

**XQuery:**

```
for $x in doc("info.xml")/dvds/dvd/title
     where doc("info.xml")/dvds/dvd/@discs = 2
          return {data($x)}
for $x in doc("info.xml")/dvds/dvd/name
     where doc("info.xml")/dvds/dvd/@discs = 2
          return {data($x)}
```

**In our approach:**

```
get_data(info, dvd, discs, 2, [name, title], Result)
```

As can be seen from the examples above, in our approach the user is not required to be familiar with the structure of the document, unlike with XPath or XQuery, where the path to the location of the information is needed. The syntax of both query languages is also more complex. This applies especially to XQuery, where the user needs to understand and know how to use variables and be somewhat familiar with coding in general.

# 9. Further development

Because this is the first prototype for a Prolog-based analysis tool, there are some suggestions for improvements that could be made to make it simpler to use and more efficient.

Currently the user has to make the queries in normal Prolog syntax, which can be somewhat complex, although the goal has been to make the queries as simple as possible. The syntax could be further simplified by using DCG (Definite Clause Grammar) [Pereira and Warren, 1980], with which the queries could be made to be more like natural language. This way the user would find the syntax more natural and less mistakes would be made. Another option would be to develop a graphical user interface, such as a form of some kind, where the user would not have to think about the actual syntax of the query, but concentrate even more on the data they want the query to return.

One direction for development could also be the addition of graphical visualization of the underlying XML source to the tool. The user could have a simple graph of the document, which would show the main elements under the root element, i.e. the next elements in the hierarchy. Clicking on an element would then expand the view to show the elements and attributes that are next in hierarchy under that element. This would give a visual tool for the user to help grasp the structure of the document. It might also be useful to provide the user with visualizations of different levels of the data which could then be used to express the user's interest in the data, for example as limits on the search.

The prototype made for this thesis can only handle one document at a time. Modifying it to handle multiple documents at one time would add the option of comparing and matching the documents. For example, if the same name appeared in several documents, all the information related to that name could be gathered and examined. Also, the user could compare the way data in different documents are presented.

An improvement to the previous could be to add functionality that would enable the user to detect data conflicts among heterogeneous data sources. Data conflicts occur when the same kind of data related to the same object is different in different data sources. For example, the deadline of some course work is different in a teacher's own notes than in the web site for the course.

The easiest way to improve the prototype would be to develop more analysis predicates. There is no limit to how many predicates there can be and thus any new analysis ideas would be useful. As the main purpose of this thesis was to show that an analysis tool can be developed in a natural way based on logical programming, only some of the more common analysis needs were considered in this prototype. A broader set of analysis predicates can be added to further develop the prototype.

There is also the way that the data from a database is returned to the Prolog program. At the moment a Prolog list is used to hold all of the data until the facts are stored in memory. However, as a list is also a kind of term, it has a limit on how much data it can hold. This might cause problems with large documents. It should be researched how much data a Prolog list can hold and if there is a better way to store the document data until it is transformed into facts.

Lastly, because the work in this thesis relates closely to dataspaces, the prototype could be used together with other dataspace tools. With some further development of the tool, it could be used in a DataSpace Support Platform to ease data management within a collection of different data sources.

## 10. Discussion

Although there are several papers on different XML query languages, there are only a few papers on how to improve the analysis of XML documents. Some want to improve an existing query language, such as XPath or XQuery, while others have come up with entirely new ways to handle XML documents. Although there are improvements to either the syntax of the queries or the traditional usage of regular path expressions, usually the need to know the structure of the document in question still remains.

The XML relation by Niemi and Järvelin that is used in this thesis is a new, alternative way to represent XML documents and to make them easier to handle and store. Brabrand et al. [2008] also talk about the need for another syntax for XML. Many XML languages already permit an alternative XML syntax because it improves the readability of the documents from the view point of the user. Using the XML relation together with the predicates developed in this thesis also helps the user make sense of the document. The user does not have to read through the document, but instead they can use the predicates to analyze the structure and content, i.e. to get knowledge about an XML document, which may be completely or partially unknown to them beforehand. This way they can first get an understanding of the document and decide whether it contains the information they are looking for or not.

In their paper [2006] Kamps et al. research using free-text queries and queries with some structural constraints. They find that three quarters of the queries they studied used some constraints on the context of the elements to be returned, which means that plain free-text queries would not be able to produce the desired results. However, Kamps et al. also found that structural constraints are not always needed at all, which suggests that a query language that can be used both with and without having to define structural constraints would be the most beneficial. They create NEXI, a query language based on a part of XPath. Kamps and his colleagues state that the reason they only use a subset of XPath is that users find it hard to specify their information needs in XPath and tend to make semantic mistakes in their query formulations. They also say that the reason for the mistakes is most likely to be that the users have no, or at best, incomplete, knowledge of the structure of the documents.

Related to the problems end-users have with XPath, the basic problem in using languages such as XQuery is that the languages rely heavily on path expressions that are based on pattern matching techniques [Näppilä et al, 2010]. They assume that the user

understands pattern matching enough to create the right kind of queries. In addition, they suppose that the user is familiar with the notion of variable of procedural languages. An XQuery query can contain several nested expressions that use variables and can contain complex interrelationships. As Näppilä et al. state, the user of these kinds of languages has to think like a programmer. End-users rarely possess these kinds of skills. The approach of this thesis, however, does not require the user to be familiar with variables. An exception to this is the notion of a shared variable, which is required if the user wants to combine different predicates to create more complex queries. The predicates can be constructed in such a way that no knowledge of variables are needed. At the same time, the predicates can be such that they can be combined together to form more complex queries if the user has the will to do so.

Although Kamps et al. design the NEXI query language for users with limited knowledge of the structure of the documents they handle, they do not take into consideration the users that have absolutely no knowledge of the documents. Using plain XPath requires the user to be aware of the document's structure. While NEXI was created to help with this, it also requires the user to have at least a basic knowledge of the structure of the XML document. This means that the user would have to browse the document or read the possible DTD or XML schema. The approach described in this paper, however, is based on the premise that the user does not need to have any prior knowledge of the documents that are being analyzed. Everything the user needs to know can be found out by applying the available analysis predicates.

An example of an attempt to move away from the path-oriented way to query XML documents is XML-GL [Ceri et al., 1999]. XML-GL is a graphical query language for XML documents, where the user formulates the query using graph-based formalism. It is not just a graphical user interface over a textual query language, but the query language itself is graph-based, and its syntax and semantics are defined in terms of graph structures and operations. XML-GL requires the documents that are being queried to have a DTD or to be well-formed, i.e. to satisfy a list of syntax rules that are provided in the XML specification. With the prototype described in this thesis, there are no similar requirements. The indexing process provides the necessary information about the structure of the document.

Another visual query language is Xing [Erwig, 2003]. The goals of the language were to not create another textual language, as well as avoiding the XML syntax. Erwig says in his paper that because the new language should be as simple as possible, avoiding the nesting pattern of XML is required. Xing does not require the document to have a DTD,

but some knowledge of the tag names is required to be able to use the language. Xing is based on pattern matching and although it supports the use of wildcards in the queries, the user has to know what tags to use or the query results can turn out to be very unrelevant.

XML-QL [Deutsch et al., 1999] is also a query language that utilizes pattern matching. The user creates a pattern of what he wants the result to look like instead of telling the query language where to look for it. This way he avoids having to use path expressions. However, as M. Erwig states in his paper [2003], to use XML-QL the user needs to know about XML syntax which again brings up the point that not many end users are necessarily very adept at reading XML nor creating the pattern the result could be found in. In addition, if using XML-QL with limited or no knowledge of the structure of the document, it requires the use of regular path expressions [Florescu et al., 2000]. For this reason one of the developers of XML-QL, Daniela Florescu, decided that the language should be extended. The motivation for the extended XML-QL [Florescu et al., 2000] was the same as the motivation for the prototype in this paper: the need for a tool or language to query XML documents with unknown structure. Florescu's example was that a user visits a (XML) website and does not know, nor want to know, how the data is stored on that website. He would still want to find some specific data easily. Florescu's answer to this is adding keyword search capabilities to the existing XML-QL. However, as discussed before, keyword searches do not always return the most relevant results.

One approach to querying semistructured data with no prior knowledge of its structure is a mechanism for implementing cooperative query processing by Barg and Wong [2003]. They had also noticed that using path-oriented query languages poses a substantial problem when the users do not know the structure of the document they are querying. Barg and Wong point out that with semistructured data it is often appropriate to return not only the exact query result but also the results that approximately match the query. The same semantic content can have vastly different structure. In these cases the user would need to know the exact path to several different locations when using XPath or XQuery.

# 11. Conclusion

The prototype developed for this thesis can not entirely replace path-oriented XML query languages because they are not made for the same purposes. XPath and XQuery are largely used in XSLT (Extensible Stylesheet Language Transformations) to locate the data that is being transformed whereas the purpose of the approach and prototype in this thesis were created to help the user to get a better understanding on XML documents which are unfamiliar with him/her beforehand.

In fact, the information gained through using the prototype in this thesis can be useful when using path-oriented languages. The end-user can utilize the prototype to find out what elements and attributes are in the document, as well as gain information about the structure. After this the usage of XPath or a similar language is much easier as the user has an idea of the structure of the underlying document. Finding out if the data the user is looking for can be found in a certain document is more easily done with the help of the prototype described in this paper than reading through the XML document or its DTD or XML schema.

Analysis of the data in an unfamiliar XML document is needed before any kind of XML manipulation or query languages can be used. All of these languages expect the user to be somewhat familiar with the document they are handling. No such assumption is made concerning the predicates described in this thesis. With the predicates the user starts examining the document from “the ground up” by just checking what the name of the root node is and moving on from there. They do not need to be familiar with programming to use the predicates but if they wish to do so, the predicates can be combined together using shared variables in the context of complex queries.

## References


[Barg and Wong, 2003] Michael Barg and Raymond K. Wong, Cooperative query answering for semistructured data. *ADC '03 Proceedings of the 14th Australasian database conference.* **17** (2003), 209-215.

[Boag et. al., 2007] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie and Jérôme Siméon, XQuery 1.0: An XML Query Language. *W3C Recommendation, available at:* http://www.w3.org/TR/2007/REC-xquery-20070123/ *(accessed 5.11.2011)*. (2007).

[Bosak et al., 1998] Jon Bosak, Tim Bray, Dan Connolly, Eve Maler, Gavin Nicol, C. Michael Sperberg-McQueen, Lauren Wood and James Clark, Guide to the W3C XML Specification ("XMLspec") DTD, Version 2.1. *Available at:* http://www.w3.org/XML/1998/06/xmlspec-report.htm *(accessed 5.11.2011)*. (1998).

[Brabrand et al., 2008] Claus Brabrand, Anders Møller and Michael I. Schwartzbach, Dual syntax for XML languages. *Information Systems*. **33**, 4-5 (2008), 385-406.

[Bray et al., 1998] Tim Bray, Jean Paoli and C. M. Sperberg-McQueen, Extensible Markup Language (XML) 1.0. *W3C Recommendation, available at:* http://www.w3.org/TR/1998/REC-xml-19980210 *(accessed 5.11.2011)*. (1998).

[Bussler, 2001] Cristoph Bussler, B2B Protocol Standards and their Role in Semantic B2B Integration Engines. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*. **24**, 1 (2001), 3-11.

[Ceri et al., 1999] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi and Letizia Tanca, XML-GL: a graphical language for querying and restructuring XML documents. *Computer Networks*. **31**, 11-16 (1999), 1171-1187.

[Chamberlin and Boyce, 1974] Donald D. Chamberlin and Raymond F. Boyce, SEQUEL: A structured English query language. *SIGFIDET '74 Proceedings of the 1974 ACM SIGFIDET workshop on Data description, access and control*. (1974), 249-264.

[Clark, 1999] James Clark, XML Transformations (XSLT). *W3C Recommendation, available at:* http://www.w3.org/TR/1999/REC-xslt-19991116 *(accessed 5.11.2011)*. (1999).

[Clark and DeRose, 1999] James Clark and Steve DeRose, XML Path Language (XPath). *W3C Recommendation, available at:* http://www.w3.org/TR/1999/REC-xpath-19991116/ *(accessed 5.11.2011)*. (1999).

[Colmerauer, 1990] Alain Colmerauer, An Introduction to Prolog III. *Communications of the ACM*. **33**, 7 (1990), 69-90.

[Deutsch et al., 1999] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy and Dan Suciu, A query language for XML. *Computer Networks*. **31**, 11-16 (1999), 1155-1169.

[EAN, 2002] EAN, IFTMIN. *Available at:* http://www.gs1.se/eancom_2002/ean02s4/user/part2/iftmin/examples.htm *(accessed 5.11.2011)*. (2002).

[Erwig, 2003] M. Erwig, Xing: a visual XML query language. *Journal of Visual Languages and Computing*. **14**, 7 (2003), 5-45.

[Fallside and Walmsley, 2004] David C. Fallside and Priscilla Walmsley, XML Schema Part 0: Primer Second Edition. *W3C Recommendation, available at:* http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/ *(accessed 5.11.2011)*. (2004).

[Fiebig et al., 2002] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkoette, Julia Neumann, Robert Till Westmann, Anatomy of a native XML base management system. *The VLDB Journal*. **11**, 4 (2002), 292-314.

[Florescu et al., 2000] Daniela Florescu, Donald Kossmann and Ioana Manolescu, Integrating keyword search into XML query processing. *Computer Networks*. **33**, 1-6 **(**2000), 119-135.

[Franklin et al., 2005] Michael Franklin, Alon Halevy and David Maier, From Databases to Dataspaces: A New Abstraction for Information Management. *ACM SIGMOD Record*. **34**, 4 (2005), 27-33.

[Gosling et al., 2005] James Gosling, Bill Joy, Guy Steele and Gilad Bracha, The Java™ Language Specification Third Edition. (2005), Addison-Wesley.

[Gou and Chirkova, 2007] Gang Gou and Rada Chirkova, Efficiently Querying Large XML Data Repositories: A Survey. *IEEE Transactions on knowledge and data engineering*. **19**, 10 (2007), 1381-1403.

[Howe et al., 2008] Bill Howe, David Maier, Nicolas Rayner and James Rucker, Quarrying Dataspaces: Schemaless Profiling of Unfamiliar Information Sources. *Proceedings of the 2008 IEEE 24th international conference on data engineering workshop*. (2008), 270-277.

[Kamps et al., 2006] Jaap Kamps, Maarten Marx, Maarten de Rijke and Börkur Sigurbjörnsson, Articulating Information Needs in XML Query Languages. *ACM Transactions on Information Systems*. **24**, 4 (2006), 407-436.

[Keogh and Davidson, 2005] James Keogh and Ken Davidson, XML Demystified. (2005), McGraw-Hill Professional Publishing.

[Klyne and Carroll, 2004] Graham Klyne and Jeremy J. Carroll, Resource Description Framework (RDF): Concepts and Abstract Syntax. *W3C Recommendation, available at:* http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/ *(accessed 5.11.2011).* (2004).

[Lu et al., 2006] Eric Jui-Lin Lu, Bo-Chan Wu and Po-Yun Chuang, An empirical study of XML data management in business information systems. *The Journal of Systems and Software*. **79**, 7 (2006), 984-1000.

[Niemi and Järvelin, 2006] Timo Niemi and Kalervo Järvelin, Another Look at XML. (2006), Tampereen Yliopistopaino Oy.

[Näppilä et al., 2010] Turkka Näppilä, Katja Moilanen and Timo Niemi, RXQL: An SQL-like Query Language for Selecting, Harmonizing, and Aggregating Data from Heterogeneous XML Data Sources. (2010), Tampereen Yliopistopaino Oy.

[Pereira and Warren, 1980] Fernando C. N. Pereira and David H. D. Warren, Definite Clause Grammars for Language analysis – A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*. **13** (1980), 231-278.

[Quass et al., 1995] D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman and J. Widom, Querying semistructured heterogeneous information. *Proceedings of Deductive and Object Oriented Databases*. (1995).

[Raggett et al., 1999] Dave Raggett, Arnaud Le Hors and Ian Jacobs, HTML 4.01 Specification. *W3C Recommendation, available at:* http://www.w3.org/TR/1999/REC-html401-19991224/ *(accessed 5.11.2011)*. (1999).

[Singleton et al., 2004] Paul Singleton, Fred Dushin and Jan Wielemaker, JPL: A bidirectional Prolog/Java interface. *Available at:* http://www.swi-prolog.org/packages/jpl/ *(accessed 5.11.2011)*. (2004).

[Tauberer, 2006] Joshua Tauberer, What is RDF. *Available at:* http://www.xml.com/pub/a/2001/01/24/rdf.html *(accessed 5.11.2011).* (2006).

[W3Schools.com] W3Schools.com: XSD How To? *Available at:* http://www.w3schools.com/schema/schema_howto.asp *(accessed 5.11.2011).*

# Appendix 1

The collection of facts for the example document in chapter 7.

```
info(dvds, e, [1])
info(dvd, e, [1, 1])
info(discs, a, [1, 1, 1])
info(1, v, [1, 1, 1, 1])
info(year, a, [1, 1, 2])
info(2005, v, [1, 1, 2, 1])
info(run_time, a, [1, 1, 3])
info(73, v, [1, 1, 3, 1])
info(title, e, [1, 1, 4])
info('Corpse', v, [1, 1, 4, 1])
info('Bride', v, [1, 1, 4, 2])
info(director, e, [1, 1, 5])
info('Tim', v, [1, 1, 5, 1])
info('Burton', v, [1, 1, 5, 2])
info(actors, e, [1, 1, 6])
info(actor, e, [1, 1, 6, 1])
info('Johnny', v, [1, 1, 6, 1, 1])
info('Depp', v, [1, 1, 6, 1, 2])
info(actor, e, [1, 1, 6, 2])
info('Helena', v, [1, 1, 6, 2, 1])
info('Bonham', v, [1, 1, 6, 2, 2])
info('Carter', v, [1, 1, 6, 2, 3])
info(dvd, e, [1, 2])
info(discs, a, [1, 2, 1])
info(2, v, [1, 2, 1, 1])
info(year, a, [1, 2, 2])
info(1984, v, [1, 2, 2, 1])
info(genre, a, [1, 2, 3])
info('Thriller', v, [1, 2, 3, 1])
info(title, e, [1, 2, 4])
info('The', v, [1, 2, 4, 1])
info('Stand', v, [1, 2, 4, 2])
info(writers, e, [1, 2, 5])
info(writer, e, [1, 2, 5, 1])
info('Stephen', v, [1, 2, 5, 1, 1])
info('King', v, [1, 2, 5, 1, 2])
info(actors, e, [1, 2, 6])
info(actor, e, [1, 2, 6, 1])
info('Gary', v, [1, 2, 6, 1, 1])
info('Sinise', v, [1, 2, 6, 1, 2])
info(actor, e, [1, 2, 6, 2])
info('Molly', v, [1, 2, 6, 2, 1])
info('Ringwald', v, [1, 2, 6, 2, 2])
info(tag_line, e, [1, 2, 7])
info(the, v, [1, 2, 7, 1])
info(end, v, [1, 2, 7, 2])
info(of, v, [1, 2, 7, 3])
info(the, v, [1, 2, 7, 4])
info(world, v, [1, 2, 7, 5])
info(is, v, [1, 2, 7, 6])
info(just, v, [1, 2, 7, 7])
info(the, v, [1, 2, 7, 8])
info(beginning, v, [1, 2, 7, 9])
info(dvd, e, [1, 3])
info(discs, a, [1, 3, 1])
info(1, v, [1, 3, 1, 1])
info(year, a, [1, 3, 2])
```

```
info(1994, v, [1, 3, 2, 1])
info(run_time, a, [1, 3, 3])
info(137, v, [1, 3, 3, 1])
info(title, e, [1, 3, 4])
info('The', v, [1, 3, 4, 1])
info('Shawshank', v, [1, 3, 4, 2])
info('Redemption', v, [1, 3, 4, 3])
info(director, e, [1, 3, 5])
info('Frank', v, [1, 3, 5, 1])
info('Dsrabont', v, [1, 3, 5, 2])
info(genre, e, [1, 3, 6])
info('Drama', v, [1, 3, 6, 1])
info(writers, e, [1, 3, 7])
info(writer, e, [1, 3, 7, 1])
info('Frank', v, [1, 3, 7, 1, 1])
info('Darabont', v, [1, 3, 7, 1, 2])
info(writer, e, [1, 3, 7, 2])
info('Stephen', v, [1, 3, 7, 2, 1])
info('King', v, [1, 3, 7, 2, 2])
info(actors, e, [1, 3, 8])
info(actor, e, [1, 3, 8, 1])
info('Tim', v, [1, 3, 8, 1, 1])
info('Robbins', v, [1, 3, 8, 1, 2])
info(actor, e, [1, 3, 8, 2])
info('Morgan', v, [1, 3, 8, 2, 1])
info('Freeman', v, [1, 3, 8, 2, 2])
info(dvd, e, [1, 4])
info(discs, a, [1, 4, 1])
info(2, v, [1, 4, 1, 1])
info(run_time, a, [1, 4, 2])
info(177, v, [1, 4, 2, 1])
info(name, e, [1, 4, 3])
info('Kamelot', v, [1, 4, 3, 1])
info('-', v, [1, 4, 3, 2])
info('One', v, [1, 4, 3, 3])
info('Cold', v, [1, 4, 3, 4])
info('Winter\'s', v, [1, 4, 3, 5])
info('Night', v, [1, 4, 3, 6])
info(genre, e, [1, 4, 4])
info('Music', v, [1, 4, 4, 1])
info(dvd, e, [1, 5])
info(discs, a, [1, 5, 1])
info(2, v, [1, 5, 1, 1])
info(run_time, a, [1, 5, 2])
info(240, v, [1, 5, 2, 1])
info(name, e, [1, 5, 3])
info('Iron', v, [1, 5, 3, 1])
info('Maiden', v, [1, 5, 3, 2])
info('-', v, [1, 5, 3, 3])
info('Live', v, [1, 5, 3, 4])
info('After', v, [1, 5, 3, 5])
info('Death', v, [1, 5, 3, 6])
info(genre, e, [1, 5, 4])
info('Music', v, [1, 5, 4, 1])
info(dvd, e, [1, 6])
info(discs, a, [1, 6, 1])
info(1, v, [1, 6, 1, 1])
info(year, a, [1, 6, 2])
info(1998, v, [1, 6, 2, 1])
info(run_time, a, [1, 6, 3])
info(115, v, [1, 6, 3, 1])
info(genre, a, [1, 6, 4])
info('Thriller', v, [1, 6, 4, 1])
info(title, e, [1, 6, 5])
info('Blade', v, [1, 6, 5, 1])
```

```
info('The', v, [1, 6, 5, 2])
info('Daywalker', v, [1, 6, 5, 3])
info(director, e, [1, 6, 6])
info('Stephen', v, [1, 6, 6, 1])
info('Norrington', v, [1, 6, 6, 2])
info(writers, e, [1, 6, 7])
info(writer, e, [1, 6, 7, 1])
info('David', v, [1, 6, 7, 1, 1])
info('S.', v, [1, 6, 7, 1, 2])
info('Goyer', v, [1, 6, 7, 1, 3])
info(actors, e, [1, 6, 8])
info(actor, e, [1, 6, 8, 1])
info('Wesley', v, [1, 6, 8, 1, 1])
info('Snipes', v, [1, 6, 8, 1, 2])
info(actor, e, [1, 6, 8, 2])
info('Stephen', v, [1, 6, 8, 2, 1])
info('Dorff', v, [1, 6, 8, 2, 2])
info(tag_line, e, [1, 6, 9])
info('It', v, [1, 6, 9, 1])
info('takes', v, [1, 6, 9, 2])
info('One', v, [1, 6, 9, 3])
info('to', v, [1, 6, 9, 4])
info('kill', v, [1, 6, 9, 5])
info('One', v, [1, 6, 9, 6])
info(dvd, e, [1, 7])
info(discs, a, [1, 7, 1])
info(1, v, [1, 7, 1, 1])
info(year, a, [1, 7, 2])
info(2008, v, [1, 7, 2, 1])
info(run_time, a, [1, 7, 3])
info(116, v, [1, 7, 3, 1])
info(genre, a, [1, 7, 4])
info('Drama', v, [1, 7, 4, 1])
info(title, e, [1, 7, 5])
info('Gran', v, [1, 7, 5, 1])
info('Torino', v, [1, 7, 5, 2])
info(director, e, [1, 7, 6])
info('Clint', v, [1, 7, 6, 1])
info('Eastwood', v, [1, 7, 6, 2])
info(actors, e, [1, 7, 7])
info(actor, e, [1, 7, 7, 1])
info('Clint', v, [1, 7, 7, 1, 1])
info('Eastwood', v, [1, 7, 7, 1, 2])
info(actor, e, [1, 7, 7, 2])
info('Bee', v, [1, 7, 7, 2, 1])
info('Vang', v, [1, 7, 7, 2, 2])
```

# Appendix 2

Sample definitions for some of the analysis predicates.

**values(Document, DataItemName, Result)**

`Values` is a predicate that gives all the values (`Result`) in the data item `DataItemName` that exist in the document `Document`.

```
values(Document, DataItemName, Result) :- tag_indices(Document,
DataItemName, IndexList), get_sub_values(Document, IndexList, Result).
values(Document, DataItemName, _) :- functor(Func, Document, 3),
arg(1, Func, DataItemName), \+Func, write('The element/attribute does
not exist.'), !.
```

The predicate takes the element or attribute name as a parameter (`DataItemName`) and returns a list of the values of all the instances of that element or attribute in the document `Document`. The function `tag_indices` is used to get a list of the indices of all elements and attributes with the name given in `DataItemName`.

```
tag_indices(Document, DataItemName, Result) :- get_findall(Document,
DataItemName, e, _, 3, IndexList1), get_findall(Document, Document, a,
_, 3, IndexList2), append(IndexList1, IndexList2, Result), !.
```

The predicate `get_findall` is used to first create lists of the indices of all attributes and elements with the given name.

```
get_findall(Document, _, Type, Index, 1, Result) :- functor(Func,
Document, 3), arg(1, Func, Component), arg(2, Func, Type), arg(3,
Func, Index), findall(Component, Func, Result).
get_findall(Document, Component, _, Index, 2, Result) :- functor(Func,
Document, 3), arg(1, Func, Component), arg(2, Func, Type), arg(3,
Func, Index), findall(Type, Func, Result).
get_findall(Document, Component, Type, _, 3, Result) :- functor(Func,
Document, 3), arg(1, Func, Component), arg(2, Func, Type), arg(3,
Func, Index), findall(Index, Func, Result).
```

The predicate takes five parameters. From these the predicate makes a functor of the form `Document(Component, Type, Index)`. Based on the fifth component, it then uses the system predicate `findall` to find all components, types or indexes in the document `Document`. The resulting list is returned in `Result`.

In `tag_indices`, `get_findall` is used to get all indices of `DataItemName` that are elements or attributes. The index lists are then combined and all of the values for those

attributes and elements are gathered into the result list which is returned to `values`. In terms of another predicate, `get_sub_values`, the final list of values is created.

```
get_sub_values(_, [], []) :- !.
get_sub_values(Document, [X|Xs], List) :- append(X, _, XList),
get_findall(Document, _, v, XList, 1, ValueList),
get_sub_values(Document, Xs, ValueList2), append(ValueList,
ValueList2, List), !.
```

`Get_sub_values` takes the document name and the index list as a parameter and returns a list of values that are found by adding one number to the given indices. The function append is used to find all possible indices. `X` is the first index in the list given as a parameter. By using _ we tell append to add anything to the index to create a new one (`XList`). After this, `get_findall` is again used to find all of the values with the newly created index. The rest of the index list, `Xs`, is used to call `get_sub_values` recursively. The recursive execution of the query ends when the first version of `get_sub_values` finds a match, or in other words, when the list of indices is empty, which would mean that all of the given indices have been handled. Finally, the value found in the first step of the process is combined to a list with the other values returned by the later `get_sub_values` call. The resulting list is returned to the user.

The second version of `values` is used only when the first one fails. It checks the existence of any fact with the given tag, `DataItemName`. System predicates `functor` and `arg` are used to do this. The predicate `functor` creates a functor named `Document` that has 3 arguments in the variable `Func`. The predicate `arg` says that the first argument in `Func` is `DataItemName`. `Func` is then used to check if any fact of the form `Document(DataItemName, _, _)` can be found. If none is found, an error message is printed and the execution of the query ends.

**get_without(Document, ElementName, DataItemName, ResultDataItems, Result)**

`Get_without` is a predicate that returns `ResultDataItems` of the `ElementName` instance in the document `Document` that does not have a data item with the name given in `DataItemName` as a parameter.

```
get_without(Document, ElementName, DataItemName, ResultDataItems,
Result) :- tag_indices(Document, DataItemName, Res1),
get_findall(Document, ElementName, e, _, 3, IndexList),
remove_matches(IndexList, Res1, Res2), show(Document, Res2,
ResultDataItems, Result).
```

First all indices for an element or attribute with the name given in DataItemName in the document Document are found with the predicate tag_indices, which was described in detail earlier. Get_findall is then used to create a list of indices of all the ElementName instances. Next another predicate, remove_matches, is used to remove all of the ElementName indices which can be found as the beginning of the indices in the list of indices for the given DataItemName. In other words, we take a list of the ElementName instances and a list of the data items. We then eliminate all ElementName instances that have the element or attribute.

```
remove_matches(List, [], List) :- !.
remove_matches(List, [X|Xs], CheckedList) :-
remove_matches_actual(List, X, CList1), remove_matches(CList1, Xs,
CheckedList).

remove_matches_actual([], _, []) :- !.
remove_matches_actual([X|Xs], Ind, Rest) :- append(X, _, Ind),
remove_matches_actual(Xs, Ind, Rest), !.
remove_matches_actual([X|Xs], Ind, Result) :- \+append(X, _, Ind),
remove_matches_actual(Xs, Ind, Res1), append([X], Res1, Result).
```

Remove_matches takes the list of indices for the named elements and the first index for the wanted attribute or element and calls the predicate that will actually remove the matches from the list of indices, remove_matches_actual. Remove_matches_actual uses the append predicate to check if the index for an element is the first part of an index for the given attribute or element. The second remove_matches_actual is for the case when a match can be found. In this case the element index is ignored and left out of the result list by calling the predicate recursively with the rest of the indices and leaving the current index out of the result list. The third remove_matches_actual instance is used for the case when no match can be found. Again the predicate is called recursively but in this case the current index is added to the rest of the results. The execution of the query ends when the list of element indices is empty. Remove_matches calls remove_matches_actual recursively also until the list of element/attribute indices is empty. When the final list of element indices is returned to get_without, the predicate show is called. The final list of indices is given as parameter (Res2) as well as the name of the document (Document) and a list of the wanted results (ResultDataItems).

```
show(_, [], _, []) :- !.
show(Document, [X|Xs], DataItem, Result) :- functor(Func, Document,
3), arg(3, Func, X), Func, show(Document, X, DataItem, Res1),
show(Document, Xs, DataItem, Res2), append([Res1], Res2, Result), !.
show(Document, Index, [DI|DataItems], Result) :- show(Document, Index,
DI, Res1), show(Document, Index, DatItems, Res2), append(Res1, Res2,
Result), !.
show(Document, Index, DataItem, []) :- functor(Func, Document, 3),
arg(1, Func, DataItem), \+Func, !.
show(Document, Index, DataItem, Result) :- append(Index, _, Index2),
functor(Func, Document, 3), arg(1, Func, DataItem), arg(3, Func,
```

```
Index2), Func, get_value_list(Document, [Index2], Result), !.
show(_, Index, DataItem, []) :- !.
```

In `show`, the first index in `Res2` (second version of predicate `show`) and the first data item in `ResultDataItems` (third version of `show`) are chosen and `show` is called recursively. The system predicates `functor` and `arg` are again used in the fourth version of `show` to create a functor of the type `Document(DataItem, _, _)`, which is used to check if a fact of that form exists. The fourth version is used when such a fact does not exist and there is nothing to return. The fifth `show` uses append to create any index for the given data item and to find the values of the data item with the predicate `get_value_list`. The last version of `show` is used when an index for the given `DataItem` cannot be found and there is nothing to return. The predicate `get_value_list` used in the fifth version of `show` is similar to the predicate `values`, that was described earlier, except it returns only values that are found under the given index, i.e. they are children of that index. It returns the values to `show`, which combines the results for each index and data item given in `Res2` and `ResultDataItems` and returns the list to `get_without`.