SEMANTIC INTEGRATION OF XML USING A GLOBAL RDF MEDIATOR

BY

FEIHONG HSU
Bachelor of Science, Mathematics and Computer Science,
University of Illinois at Urbana-Champaign, Urbana, Illinois, 2000

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2004

Chicago, Illinois

To my parents and my sister...

ACKNOWLEDGMENTS

I would like to thank Prof. Isabel Cruz for providing much-needed guidance and feedback. Without your help I would not only have been lost, I would have never even known where to start.

Thanks also to Huiyong Xiao, with whom I had numerous, valuable discussions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	eXtensible Markup Language
RDF	Resource Description Framework
DTD	Document Type Definition
Relax NG	Regular Language description for XML
DAML+OIL	Darpa Agent Markup Language + Ontology Inference Layer
OWL	Ontology Web Language
RDFS	RDF Schema
XPath	XML Path Language
XQuery	XML Query Language
RDQL	RDF Data Query Language
RQL	RDF Query Language
SeRQL	Sesame RDF Query Language

SUMMARY

XML is one of the most important languages on the Web, especially for data interchange. Because of the proliferation of heterogeneous XML documents, a lot of recent work has been done on the semantic integration of XML documents. Previous efforts for XML semantic integration have focused on shoehorning XQuery or creating custom languages. In our approach, we choose to use RDF in a layered, mediated framework. Our framework incorporates a global ontology (expressed in RDF Schema) that provides a view of the local XML schemas. The global mediator accepts RDQL queries and translates them to the appropriate XQuery expressions, one for each local data source. The results of the queries are manipulated and presented in a way that will allow for almost seamless integration with RDF. A partial implementation of the framework has been carried out.

CHAPTER 1

INTRODUCTION

Today, much of the information on the Web is represented in XML, or can easily be converted into an XML-based format. XML has had enormous success in becoming a de-facto language for data interchange, serialization, RPC, and other practical, real-world applications. The question that often faces developers is not why they should use XML but rather how they can take advantage of its power.

As more and more developers use XML, the Web has seen a great proliferation of XML documents. These documents contain all sorts of useful information from many different sources and application domains. It would be desirable if the information content from different XML documents could be integrated to allow them to be processed in a uniform way—this is often called *data integration*. Data integration attempts to overcome the schematic differences of XML documents; this in turn can allow a single query to be processed with respect to multiple data sources, each source conforming to its own schema. However, data integration merely provides a uniform interface to heterogeneous data, whereas we are also interested in combining information from the distinct data sources, and querying at a higher level of abstraction. We can achieve these goals using *semantic integration*, which deals not only with schematic heterogeneity, but also with semantic heterogeneity.

Once semantic integration is achieved, it can be beneficial in many application areas: Perhaps the XML streams from two different sensors could be integrated to provide more accurate or precise data (or even to derive new data). Or maybe the XML data from a scheduling calen-

dar and an address book could be combined to offer richer functionality to the user. Another possible application is in the area of software agents—programs which search the Web for useful information. Semantic integration would allow software agents to perform searches on a much wider variety of documents.

1.1 Related Work

Many people in the research community have been hard at work on data integration and semantic integration projects for some time now. In this section we provide a brief, select survey of such projects. We also take some time to compare and contrast the different approaches with respect to our own framework.

1.1.1 Data Integration Approaches

There are a number of approaches that focus on data integration of XML, ignoring the semantic aspect of integration. These projects deal only with schematic heterogeneity of XML documents. One such project is CXQuery (6), an XML language used for querying and updating. Despite its name, it has less in common with XQuery and more in common with constraint query languages like Datalog. However, it does make minimal use of the XPath subset of XQuery. CXQuery can be used to integrate XML documents with different schemas by first transforming them into instances of a global schema. Both global and local schemas are expressed in DTD, a simple XML schema language. This approach is far removed from ours as we use a global ontology, not a structure-based global schema. However, we do convert some of the data from local sources to data that the global mediator can directly work with.

Piazza (8) is a peer-to-peer system introduced by Halevy et al. that enables interoperability between web resources. Although the authors claim that Piazza can enable interoperability

between RDF and XML sources, they mainly focus on interoperability between heterogeneous XML sources. Because their work does not explicitly deal with ontologies, we are reluctant to classify Piazza as a semantic integration system. Also, Piazza is a peer-to-peer system and so has no global mediator. Data integration is achieved via a query answering algorithm that maps XQuery expressions from one source to XQuery expressions for another source, taking into account the different schemas of the sources. The language used to describe the mappings between schemas is a limited form of XQuery. The query answering algorithm is based on pattern matching, which is also how we handle query translation in our integration framework. However, we do not use pattern matching of trees.

1.1.2   Semantic Integration Approaches

Semantic integration generally involves a conceptual query language and a semistructured query language. The conceptual query language is understood by the global ontology while the semistructured query language is used to extract information from local data sources. Integration is achieved when the global mediator translates conceptual queries into heterogeneous semistructured queries that are sent to multiple, heterogeneous data sources. Most of the approaches follow this basic pattern.

Klein describes an approach that can semantically annotate XML documents (9). In it, he uses external RDF documents and an ontology expressed in RDF Schema to describe, at the conceptual level, the contents of XML documents. This work is a not a framework for semantic integration per se, but it does affirm the usefulness of ontologies when dealing with semantic integration. Also notable is Klein's use of standardized Semantic Web languages like RDF and RDF Schema instead of custom languages.

Camillo et al. propose a conceptual query language called CXPath (5). CXPath, as its name suggests, is syntactically similar to XPath. This is an advantageous because the local sources are queried using XPath, and CXPath allows for relatively easy translation to XPath. However, CXPath's path expressions are different from XPath's because they describe paths in a graph, not paths in a tree. The conceptual model for the global mediator is ORM/NIAM, a variant of the ER model. Translation from CXPath to XPath is achieved by using a mapping views approach. In contrast, our framework uses XQuery as the query language for the local XML data sources, which makes translation more difficult but also allows for more expressive queries. Also, Camillo et al. do not explain how the results from the XPath queries are translated into a usable form.

Another interesting approach is that undertaken by Amann et al (1). They use OQL (10) as their conceptual query language and XQuery as their semistructured language. They use a generic conceptual model that resembles RDF (but in fact could just as easily be ER). Mapping between OQL and XQuery is done via a set of mapping rules. The biggest difference between their approach and ours is that their mapping rules are more expressive than our mapping structures; in fact they can be interpreted as statements about the implicit semantics of the local XML documents. This expressiveness does, however, make their translation algorithm more complicated. Also, like Camillo et al. they do not address the transformation of results, and their conceptual query language, OQL, is not a Semantic Web language.

Vdovjak and Houben describe an architecture which involves a global RDF mediator that integrates heterogeneous XML sources (20). This describes our project in a nutshell. However, their work focuses on a high-level overview, and they do not specify which query languages and

which schema languages to use. Also, they tout the benefits of a layered architecture, which facilitates modularization. Modularization, in turn, makes the individual parts of the system easier to design.

Lakshmanan and Sadri propose an infrastructure for XML data interoperability based on the use of ontologies (14). They use a generic conceptual model but their global mediator uses XQuery just as the local sources do. This is because the semantic relationships are embedded in the XQuery expressions used by the global mediator. However, because they use a language not specifically designed for ontologies, they do not benefit from the expressiveness of conceptual query languages, i.e. inheritance relationships. In contrast to Heuser et al., they use a mapping catalog to handle query translation, which is also how we handle mapping.

Finally, Patel-Schneider and Siméon propose the Yin/Yang Web, a framework that attempts to integrate XML and RDF through a unified model (15). This unified model tries to bridge the gap between XML and RDF by defining semantic interpretations of XML documents (this is quite different from our mapping catalogs approach, but quite similar to the work of Amann et al.). They use RDF Schema for their conceptual model but like Lakshmanan and Sadri they use XQuery as their conceptual query language. XQuery was never conceived as a conceptual query language, so they sidestep this limitation by using XQuery to query their semantic interpretations, thereby indirectly translating the query to a form that the local source can understand. However, by using XQuery they assure that even simple queries to the global ontology are hard to write and also to comprehend. In addition, they lose the clean separation between the global ontology and the local data sources. In fairness, the authors do claim that

their approach would allow the use of RQL, a query language specifically designed for RDF, but due to space considerations they do not elaborate on how this can be done.

1.2   Contribution

We present a layered framework for the semantic integration of XML documents. This framework follows the Global-As-View paradigm, in which a single global schema provides a view over multiple local schemas. In our semantic integration framework, the global schema is an ontology. Other approaches to XML semantic integration have tried to either adapt XQuery to this purpose, or to use custom ontology languages. We, in contrast, choose RDF Schema to represent our global ontology. Unlike some other ontology languages, RDF Schema has been developed and standardized by the W3C. Therefore it has been designed expressly for the Semantic Web, and there are currently many tools and libraries for processing RDF and RDF Schema. In addition, RDF can interoperate with higher-level ontology languages such as DAML+OIL and OWL. Our layered framework can take advantage of this interoperability by allowing higher layers to be added above the RDF Schema layer.

In our framework, queries to the global mediator are handled via RDQL, a standardized language for querying RDF models. The mediator translates RDQL into XQuery (an XML query language) with the help of mapping structures. Each mapping structures is associated with one XML data source and dictates how RDF triples can be translated to XML path expressions. Once the XQueries have finished executing on the XML sources, their results are passed back to the mediator, which aggregates and formats the results in a way that RDQL expects.

Whereas many approaches for semantic integration tend to focus on just query translation, our approach addresses the issues of both query translation and result transformation. Therefore it covers the entire process by which the user enters a query and receives a result in a usable format. In this sense it is the most complete integration framework to date. However, due to the overalll complexity of semantic integration we must gloss over some of the finer-grain issues. For instance, we do not propose any new or novel methods for the semi-automatic generation of mapping structures. We also do not attempt to explain how local schemas can be semi-automatically merged into the global schema. In conclusion, the main contribution of this document is the introduction of a practical framework for XML semantic integration based on RDF. It is practical because the integrated system simulates an RDF model and can therefore interoperate with real RDF models and RDF processors. Also, our use of standardized languages and easily-obtainable tools makes the implementation easier and more straight-forward.

CHAPTER 2

SEMANTIC WEB TECHNOLOGIES

Our framework relies on a number of Semantic Web technologies. The most important technologies we use are XML- and RDF-related languages. Most of the languages we present here have been standardized, and there are freely-available implementations for all of them. The languages fall into three main categories: data languages, schema languages, and query languages.

2.1 Data Languges

Data languages are languages which can be used to encode data. The only data languages we are concerned with are XML and RDF. In the integrated system, XML is the language used for the local data sources, and RDF is the language used by the global mediator.

2.1.1 XML (eXtensible Markup Language)

XML is an important language for data interchange, and also provides a serialization format for Semantic Web languages. Like HTML, it is derived from SGML, but is far less complex than its parent language. Unlike HTML, however, XML has a stricter language standard. All XML documents must have a hierarchical structure to be called well-formed, meaning that well-formed documents can be represented as trees. Well-formed XML documents can easily be parsed and processed by a wide array of tools and libaries. A great number of languages have an XML-based syntax, for example XSLT, RDF, SVG, Ant, Jelly, and XHTML. These languages are used in many different application domains, some of them not even Web-related, and their existence reflects the flexibility of XML.

XML documents consist of elements, attributes, and text (just as in HTML). Elements are nodes in the XML tree which are denoted by tags. For example, ⟨item⟩ is a tag whose name is "item". Elements can have child nodes, and these child nodes can either be other elements or plain text. Attributes are key-value pairs that occur in the context of an element. To understand attributes, it is best to think of each element as having a dictionary, with each attribute being an entry in that dictionary. For example, ⟨itemid="32X-37G"weight="45kg"⟩ denotes an element named item that has two attributes, id and weight, with values "32X-37G" and "45 kg", respectively.

2.1.2 RDF (Resource Description Framework)

RDF is not strictly a data language. It was originally designed to describe resources on the Web, so it is really more of a metadata language. However, anything that can be encoded in XML can also be encoded in RDF, although perhaps not as compactly. While XML has a tree structure, RDF has a conceptual model which is a graph. The nodes in the graph are called resources (in RDF parlance). When dealing with RDF, one does not think in terms of documents, but rather, models. The information in RDF models can be distributed in several places across the web. This is because each resource in the RDF model (graph) is identified by a URI (Uniform Resource Identifier). The URI usually takes the form of a URL (Uniform Resource Locator), a unique address on the Web. Therefore, resources can be uniquely identified even in the context of the Web. For example, a document d1.rdf and another document d2.rdf are part of the same model if they both reference the resource located at http://example.org/Hermes. Figure 1 shows an RDF model represented as a graph.
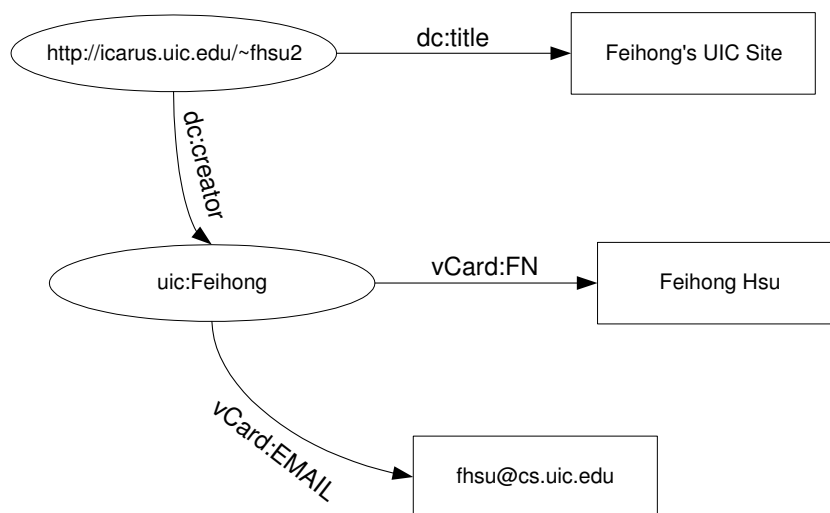
Figure 1: An RDF model in graph form

Although RDF models should always be thought of as graphs, it is useful to have a standard construct for describing parts of the graph. For this purpose, RDF provides the triple: a statement which consists of a subject, a predicate, and an object. A triple resembles a simple English sentence that contains only a noun, a verb, and the object of the verb. For instance, the RDF triple

```
<http://icarus.uic.edu/~fhsu> dc:creator uic:Feihong .
```

has subject ⟨`http://icarus.uic.edu/~fhsu`⟩, predicate `dc:creator`, and object `uic:Feihong`. Semantically, this statement probably means that the web site at ⟨`http://icarus.uic.edu/~fhsu`⟩ was created by a person identified as `uic:Feihong`. Note that all three parts of the triple are URIs, even though it may not seem like it. The `dc:creator` and `uic:Feihong` elements are abbreviated URIs, often used by RDF authors to make their models more read-

able. The URI `dc:creator` consists of two parts: the namespace prefix, `dc`, and the local name, `creator`. The namespace prefix `dc` is just a shorthand for the namespace `http://purl.org/dc/elements/1.1/`. When the namespace prefix is expanded the full URI is actually `http://purl.org/dc/elements/1.1/creator`. The mappings between prefixes and namespaces are typically defined by an author at the top of an RDF document.

Seen from a graph-centric perspective, RDF triples define an arc in the RDF graph. This means that the subject and objects of the triple are source and target nodes for a labelled edge defined by the predicate. For example, the triple shown above could be represented in graph form as:
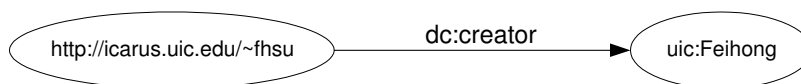
The two nodes are resources, and the unidirectional edge is called a *property*. Properties in RDF relate resources to each other. Resources can be URIs, but there is another kind of resource: the *literal*, which is essentially a string constant. The string constant can actually be in XML or some other language, but to RDF they are all the same because RDF does not have much support for data typing. Literals can never appear as the subject of a triple; they can only fill the object slot of the triple. Below we have a triple containing a literal. Note that literals in the graph are represented with rectangles, not ellipses.
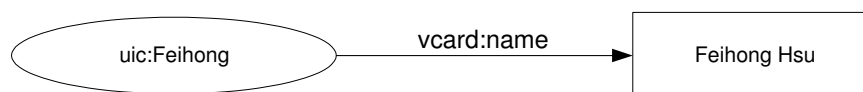
Although RDF has a graph structure, it is possible to view an RDF model as an unordered collection of triples. Seen in this way, RDF is both flatter and more open-ended than XML. All

one has to do to extend an RDF model is add more triples to it (this is equivalent to drawing new arcs on the graph).

Like XML, other languages can be built on top of RDF. One of the most important RDF-based languages is RDF Schema, the schema language for RDF. RDF Schema is expressed in RDF, though it is used for a different purpose than RDF. We will discuss RDF Schema more in a later section.

There are a few different ways of encoding RDF models using text. One text-based RDF language is RDF/XML, an XML serialization for RDF. RDF/XML is fairly flexible and allows several ways of specifying the same model; therefore an RDF/XML document does not always have a flat structure. There is also a notation called N-Triple, which is simply a flat listing of the triples contained in a given document. The most readable format for representing RDF models is N3 (Notation 3) (23). N3 is somewhat similar to RDF/XML in that it allows various shorthand notations which make the document more compact. However, N3 is more readable because it does not use XML's tag-based syntax. For this reason, we will use N3 in all of our RDF examples. Figure 2 shows the same RDF model as in Figure 1, except that in Figure 2 the model is represented in N3.

2.2 Schema Languages

A schema is something that describes a document or model; in essence, a schema provides metadata. Schema languages provide a way of encoding schemas; however, the role of a schema language depends on the context in which it is applied. A schema for XML provides the grammar for its instance documents, thus defining the hierarchical structure as well as the

```
@prefix uic:        <http://www.cs.uic.edu/people#> .
@prefix dc:         <http://purl.org/dc/elements/1.1/> .
@prefix vCard:      <http://www.w3.org/2001/vcard-rdf/3.0#> .

<http://icarus.uic.edu/~fhsu2>
    dc:creator uic:Feihong;
    dc:title "Feihong's UIC Site" .

uic:Feihong
    vCard:FN "Feihong Hsu";
    vCard:EMAIL "fhsu@cs.uic.edu" .
```

Figure 2: An RDF model represented in Notation 3

names of its nodes. An XML document is considered valid with respect to a schema if it satisfies all the constraints in the schema.

In contrast, a schema for RDF describes the classes and properties that can be found in an RDF model. Unlike XML, there is no issue of conformance, since an RDF model does not have to contain instances of all classes and properties defined in a given RDF schema. Rather, RDF schemas provide rules or facts that can be used as a basis for inferencing. The main schema languages for XML are W3C XML Schema and DTD. We will also discuss an interesting XML schema language called RelaxNG. The only schema language for RDF is RDF Schema, but we will briefly cover DAML+OIL and OWL, which are two ontology languages built on top of RDF Schema.

2.2.1 DTD (Document Type Definition)

DTD (24) is the earliest schema language developed for XML. Like all XML schema languages, DTD constrains instance documents by defining the names of elements and attributes along with their relative positions in the nested structure. Unlike later XML schema languages, DTD does not have an XML serialization (though DTDs can be embedded inside XML documents).

2.2.2 W3C XML Schema

XML Schema (7) is the schema language designed to replace DTD. Despite its title, it is not the only modern schema language available for XML. Often, XML Schema is referred to as W3C XML Schema, since this name is less ambiguous and since W3C is the group that standardized it. Unlike DTD, XML Schema is expressed in XML. XML Schema is more powerful than DTD, having facilities for specifying the following additional constraints: data type, formatting, and fine-grained cardinality. XML Schema is gaining prominence, but many research papers still use DTD to present their examples. XML Schema has often been criticized for being too verbose and unnecessarily complex. For this reason, we will refrain from using XML Schema in our examples, while noting that, in principle, our framework can be made to work with XML Schema (or at least a significant subset).

2.2.3 Relax NG (Regular Language description for XML)

Relax NG (22) is a modern XML schema language that combines the best features of DTD and W3C XML Schema. Like DTD, it has a very simple syntax, but it also possesses most of the important features of XML Schema. There are two valid syntaxes for Relax NG: the compact syntax, which, like DTD, is not expressed in XML; and an XML-based syntax. Since

```
element employees {
  element employee {
    attribute id { xsd:integer },
    attribute name { text }
    attribute position { text },
    element tasks {
      element task {
        attribute name { text }
        attribute desc { text }
      }+
    }?
  }+
}
```

Figure 3: An XML schema written in Relax NG

the compact syntax of RelaxNG is more concise, readable, and expressive than DTD, we will use RelaxNG in all of our examples. Figure 3 shows a typical XML schema written in Relax NG; this schema describes the grammar for documents which contain information on employees, their personal data, and the tasks they are assigned to. In Figure 4 we have an XML document which conforms to the XML schema in Figure 3.

2.2.4 RDFS (RDF Schema)

RDFS (4) is based on RDF and provides a vocabulary to describe classes and properties (also known as concepts and roles). In fact, RDFS ontologies are written in RDF, but RDFS ontologies should not necessarily be treated as RDF models. One key difference of RDFS that both classes and properties can be nodes in the graph, whereas properties are always edges in RDF. Inheritance of classes is supported and multiple inheritance is allowed. Unlike the object-oriented paradigm, properties can also be inherited in much the same way as classes.

```
<?xml version="1.0"?>

  <employees>
  <employee id="104" name="Humphrey Dinker" position="Manager"/>

  <employee id="105" name="Dorothy Stewart" position="Clerk">
    <tasks>
      <task name="sort" desc="Sort all letters into bins"/>
    </tasks>
  </employee>

  <employee id="105" name="Skyler Navi" position="Intern">
    <tasks>
      <task name="coffee" desc="Get coffee for everyone"/>
      <task name="lunch" desc="Get lunches for everyone"/>
      <task name="pencil" desc="Sharpen everyone's pencils"/>
    </tasks>
  </employee>
</employees>
```

Figure 4: An XML document that conforms to the Relax NG schema in Figure 3

In addition, one can define the domain and range of a property, which constrains the kind of resources that can appear in a triple that uses the property in the predicate slot. The domain constrains the subject, and the range constrains the object. For example, if there is a property named `paints` with domain `Painter` and range `Painting`. Then a triple with `paints` in the predicate slot will only allow instances of `Painter` in the subject slot, and it will only allow instances of `Painting` in the object slot.

Unlike XML schema languages, RDFS is not used to define or constrain nested structure and formatting. Rather, an RDFS ontology contains semantic metadata which allows inferencing to take place on RDF models. This, in turn, allows more interesting queries on the model to be answered. For example, if `Painter` and `Novelist` are classes, and both are subclasses of

`Artist` (another class), we can define a query that asks for all the `Artist` instances in the model. An RDF model without a rule engine would only return objects that are known to be instances of `Artist`. A rule engine that understands RDFS could figure out that `Painter` and `Novelist` instances should also be returned. This is similar to the notion of polymorphism in the object-oriented paradigm. In our framework, RDFS is used to define the global ontology. This global ontology provides a view over the local XML data sources.

2.2.5 DAML+OIL (Darpa Agent Markup Language + Ontology Inference Layer)

DAML+OIL (16), as its name suggests, is the result of merging two languages into one. DAML+OIL belongs in the class of languages known as Description Logic. Since it is built on top of RDF Schema, DAML+OIL ontologies and RDF models can seamlessly interoperate with each other. That means that an RDF model with DAML+OIL elements in it can still be understood and processed by RDF tools (though RDF tools, by themselves, will not be able to perform description logic inferences). We mention DAML+OIL here because it can be used to build a higher-level layer for our layered semantic integration framework.

2.2.6 OWL (Ontology Web Language)

OWL (17) is very similar to DAML+OIL, plus some additional constructs (and minus some constructs). In the tradition of Description Logic languages, OWL has some variants (or subsets) which offer different degress of computational complexity and expressiveness. For example, OWL Lite has the best computational properties while offering the least expressiveness, and OWL Full is very expressive but relatively expensive. Like DAML+OIL, OWL can interoperate with RDF models, and so can be used to build a higher-level layer for our framework.

2.3   Query Languages

Query languages are used to search for information in documents or models. XPath and XQuery are the most common query languages for XML, although XPath is really a subset of XQuery. In our framework, XQuery is used to query the local data sources. For RDF, there are three query languages: RDQL, RQL, and SeRQL. Although our global mediator only understands RDQL, the three query languages are actually quite similar, so in the future we may support all three.

2.3.1   XPath (XML Path)

Since XML has a tree structure, we can use paths to specify nodes in the tree. XPath (2) is a language used for specifying such paths. XPath consists of path expressions, which are similar to paths in a file system (e.g. `C:\ProgramFiles\Java\`), but are more complex because they may contain operators and filters. A simple XPath expression would be:

```
/employees/employee/name/given
```

This XPath expression would return all the nodes in the target document that can be found in that particular path. A more complex XPath expression would be:

```
/employees/employee[5]/name/given[text() = "Mike"]
```

This XPath expression contains two predicate filters, each of which puts further constraints on what node(s) can be returned. The first predicate filter specifies that the `employee` node must be the fifth node under the `employees` node; the second predicate filter specifies that the given node must contain text that matches the string "Mike". Only nodes that satisfy all conditions specified by the predicate filters can be returned. XPath figures heavily in our framework, where it is used both in the mapping structures and in URIs. The XPath expressions that

appear in our framework will be simple ones that do not contain any predicate filters exception `position()`.

2.3.2 XQuery (XML Query)

XQuery (3) is a major query language for XML. It is a much more powerful than XPath for searching XML documents, and in fact is a superset of XPath. XQuery resembles a functional programming language that uses the FLWR syntax. FLWR languages have four main clauses: `for`, `let`, `where` and `return`. The `for` clause is a looping construct that iterates through the elements of a sequence. The `let` clause is used to bind variables to the results of expressions. The `where` clause is used to specify boolean expressions which constrain the search. The `return` clause is similar to the `SELECT` clause in SQL—it specifies which nodes or expressions will be returned to the user. There is another clause, `orderby`, which we do not use and will not discuss further.

Like SQL, XQuery expressions can be nested. Unlike SQL, the results of an XQuery are not returned as a table but rather as a *DOM tree*. DOM stands for Document Object Model, and is a standardized API for manipulating and processing XML documents (25). There are numerous implementations of DOM for every major programming language. A DOM tree is essentially a tree data structure whose nodes correspond to the elements, attributes, and text contained in an XML document. Each node has methods to access its parent node and its child nodes (if it has any). In our framework, the local XML data sources understand XQuery and return the results of XQueries as DOM trees.

2.3.3   RDQL (RDF Data Query Language)

RDQL (11) is a query language that extracts data from RDF graphs. It is based on SQL so it returns tables of RDF resources instead of RDF itself. RDQL has the same basic clauses as SQL: `SELECT`, `FROM`, and `WHERE`; there are also the additional clauses `AND` and `USING`. The `SELECT` clause indicates which variables will be returned in the table (the number of variables in the `SELECT` clause is equal to the number of columns in the result table). The `FROM` clause specifies the model(s) which will be queried; if left out, it is assumed that the query will be executed against a model loaded in memory. The `WHERE` clause is essentially a list of triples, where some of the triple elements are variables. Since RDF models are also collections of triples, the `WHERE` clause is actually a parameterized subgraph. If the parameterized subgraph in the `WHERE` clause can be "found" in the model, the variables specified in the `SELECT` clause will be added to the result table. The `AND` clause is used to specify constraints using boolean expressions. It works identically to SQL's `WHERE` clause. The `USING` clause is essentially a list of mappings which allows the URIs in the `WHERE` clause to be abbreviated; it maps prefixes to the namespaces that they expand to. RDQL queries are accepted by the global mediator in our integration framework, which returns the results as tables of resources.

The result of an RDQL query, as previously stated, is a table. This table contains resources, or, more specifically, the URIs of resources. If the resource is a literal, a string constant is supplied instead of a URI. Although RDQL does not really return RDF triples, it is possible to simulate the effect by returning a three-column table, where the first, second, and third columns represent the subject, predicate, and object, respectively.

2.3.4   RQL (RDF Query Language)

RQL (12) is a query language for RDF that is also based on SQL. Its syntax is somewhat different from that of RDQL; the most notable difference being that RQL allows arbitrarily long path expressions in the `WHERE` clause. RDF triples can be considered to be path expressions of length two, but RQL allows path expressions of any length. This makes some queries more concise and readable. We do not use RQL in our integration framework because RDQL is easier to translate to XQuery. However, future implementations may support RQL.

2.3.5   SeRQL (Sesame RDF Query Language)

SeRQL (13) is based on RQL but introduces an important innovation for an RDF query language: the ability to return RDF models instead of tables. SeRQL has two kinds of queries: select queries and construct queries. Select queries are very similar to the queries used in both RDQL and RQL. However, construct queries use a new CONSTRUCT clause and return RDF models. SeRQL is, in some ways, a better fit for the integration framework than RDQL. However, RDQL is much easier to work with in terms of implementation. Some future work should be done to allow the global mediator to understand SeRQL queries.

CHAPTER 3

KEY PROBLEMS OF SEMANTIC INTEGRATION

There are four major issues that stand in the way of semantic integration: schematic heterogeneity, semantic heterogeneity, semantic relationships, and object identity. In the following sections we will go over each issue in some detail. The reader should keep in mind that these are but a few of the many potential problems in semantic integration. Since RDF Schema is a low-level ontology language, there are many issues that it cannot handle. Therefore we limit ourselves to what we feel are the most important and relevant problems.

3.1   Schematic Heterogeneity

Schematic heterogeneity exists when different documents conform to different schemas. In XML, this usually means that one document has a different nesting structure than another document. Even if two XML documents have the same information content but are instances of two different schemas, they can have completely different nestings. Thus, querying a collection of XML documents can be difficult and tedious, since meaningful XML queries must be written with knowledge of a particular documents nested structure.

For example, consider two XML documents, `books.xml` and `authors.xml` (shown in Figure 5). These two documents have the same information content but they differ in one important aspect: the nested structure of `books.xml` is the inverse of the nested structure of `authors.xml`. To get a list of book titles and their respective authors, one must write two mutually-incompatible queries, Q_{author} and Q_{book} (also shown in Figure 5). That is, Q_{author} will not work on `books.xml`, and Q_{book} will not work on `authors.xml`. Now imagine that there

```
                                        <?xml version="1.0"?>

    <?xml version="1.0"?>                <books>
                                          <book title="Semantic Integration">
    <authors>                              <author name="Huiyong Xiao"/>
      <author name="Huiyong Xiao">        </book>
        <book title="Semantic Integration"/>  <book title="Python Unlimited">
        <book title="Java Squared"/>          <author name="Flora Wu"/>
      </author>                            </book>
      <author name="Flora Wu">             <book title="Java Squared">
        <book title="Python Unlimited"/>       <author name="Flora Wu"/>
        <book title="Java Squared"/>           <author name="Huiyong Xiao"/>
      </author>                            </book>
    </authors>                           </books>
```

(a) authors.xml

(b) books.xml

```
for $author in doc("authors.xml")/authors/author
for $book in $author/book
return ($book/@title, $author/@name)
```

(c) Q_{author}

```
for $book in doc("books.xml")/books/book
return ($book/@title, $book/author/@name)
```

(d) Q_{book}

Figure 5: Two XML documents and their respective XQueries

are actually 20 such XML documents containing information on books and their authors. If each document has a different schema, a developer must write 20 different queries to extract the same type of information from each one! This is clearly an undesirable situation, and it would be much better if a single query could be used to search all the documents.

Semantic integration removes the need for nested queries by taking advantage of the more flexible nature of RDF. Consider the equivalent query in RDQL (for fetching book titles and their associated authors), shown in Figure 6.

```
SELECT ?title, ?author
WHERE (?book, <book:title>, ?title),
      (?author, <book:writes>, ?book)
```

Figure 6: An RDQL query that requests book titles and author names

The RDQL query uses RDF triples and is more readable and comprehensible than the two XQuery expressions we showed. This is because it does not have to deal with the structure of data; it only deals with the semantics of the data. Of course, processing the RDQL query can be much more expensive than processing the XQuery expression, but such is the price one must pay for higher levels of abstraction.

3.2   Semantic Heterogeneity

Every XML document has an implicit conceptual model. This conceptual model, or ontology, contains the classes and properties that describe an XML document at the semantic level. However, this conceptual model only exists in the mind of the XML document's author—it cannot be captured by any XML schema language. XML schema languages only provide descriptive metadata, which gives names to the elements and attributes that can appear in a document. Although the names of elements and attributes in an XML document give clues to their meaning, these names are generally too ambiguous. Names alone are insufficient to interpret a document at the semantic level because of synonymy and polysemy (19). Synonymy is a problem because different names can be used as labels for the same concept. For example, film and movie are two different names for the same thing, but an XML document that has element `film` probably has a different schema than an XML document that has element `movie`.

On the other hand, polysemy is a problem because a single name can have different meanings (depending on the context). For example, the name star can either mean a "gaseous heavenly body" or a "prominent, highly-paid actor". Since disambiguating the names in an XML document is nontrivial, we cannot say that XML documents have any sort of explicit semantics. In addition, the implicit semantics of distinct XML documents can differ significantly from each other.

3.3 <u>Semantic Relationships</u>

The implicit conceptual model of an XML document might contain classes or properties that do not directly correspond to any elements or attributes in the document. This can occur if hypernyms and hyponyms are present. Hypernyms are generalizations of words, and in RDF they are referred to as superclasses. For example, performer is a hypernym of actor (since performer can include singers, dancers, etc. as well as actors). Hyponyms, on the other hand, are specializations of words; in RDF they are known as subclasses. For example, poodle is a hyponym of dog (since a poodle is a specific kind of dog). Semantic relationships like hypernymy and hyponymy can be beneficial if they can be used in conceptual queries. For instance, say that there is an XML document, `trucks.xml`, which contains information on trucks, and that there is another XML document, `planes.xml`, which contains information on planes. Using XQuery, there is no way to send a query that asks for information on all vehicles (vehicle being a hypernym of both trucks and planes). However, a conceptual query, perhaps expressed in RDQL, could perform this query and get meaningful results from both `trucks.xml` and `planes.xml` (assuming that semantic integration has been achieved).

Hypernyms and hyponym relationships apply for nouns like plane, truck, and vehicle, but they also apply for verbs like pilot, drive, and control (piloting and driving are two different ways of controlling a vehicle). Therefore, a conceptual query could ask for information on people and the vehicles they control. This query would be impossible to formulate in XQuery, because the XQuery expressions are too tied to the structure and the names of elements in an XML document. If the name vehicle or the name control does not explicitly appear in the document, XQuery will not return any results at all.

3.4   Object Identity

RDF resources have unique identifiers in the form of URIs. Even a so-called anonymous resource (also called a *bnode*), has a unique identifier (although it may be randomly generated). This allows information about a resource to be distributed across the Web. For instance, there might be a resource whose URI is ⟨`http://example.org/people#Mike`⟩. Facts about this resource (triples containing the URI for the resource) might appear in a document at ⟨`http://thyme.net/peole.rdf`⟩ or a document at ⟨`http://oregano.info/people.rdf`⟩. It does not matter where facts about the resource are kept, because the URI of the resource allows it be referenced in an unambiguous way.

URIs do help to identify resources, but they are not the only way. Sometimes, the properties associated with a resource can be used to identify it. For example, a `Book` resource (resource of type `Book`) that has an ISBN of 0596002815 is equivalent to another `Book` resource whose ISBN is 0596002815. We can say that ISBN is a functional property—the value of the property's object is enough to identify its subject. In terms of our example, it means that every book

has a unique ISBN, so `Book` resources that have the same ISBN must really represent the same

object. In terms of RDF, we might write these facts as:

```
amazon:LearnPy book:isbn "0596002815" .
oreilly:LearningPython book:isbn "0596002815" .
amazon:LearnPy owl:sameIndividualAs oreilly:LearningPython .
```

Note the third triple, which states that the first Book resource is the same individual (object)

as the second `Book` resource. The prefix we used in that triple had the OWL prefix because

RDF and RDF Schema do not have facilities for saying that two resources are the same object.

However, this issue of object identity is important in semantic integration because it allows

information from different sources to be shared. For example, given the following triples

```
<amazon:LearnPy>, <book:publishedIn>, "December 2003"
<oreilly:LearningPython>, <book:title>, "Learning Python"
```

we can now conclude that a `Book` resource whose ISBN is 0596002815 has the title "Learning

Python" and was published in December 2003. Without being able to state that ⟨amazon:

LearnPy⟩ and ⟨oreilly:LearningPython⟩ correspond to the same object, we could only con-

clude that there is one book named "Learning Python" and that there is another book that

was published in December 2003.

Much as object identity is a problem in RDF, it is an even bigger problem in XML. The

elements and attributes of an XML document generally do not have any kind of unique identifier

associated with them. Likewise, the nodes in a DOM tree do not contain any identifying

information other than the name of the node, which is not very useful because many nodes in

the same tree can have the same name. Therefore the node in one tree may contain the same

information as a node in another tree, but unlike RDF there is no way to specify that the two

nodes are related. Furthermore, since XML documents and DOM trees do not contain semantic

information, there is often no easy way to find equivalence between nodes. For example, a `book` element in one document may be semantically related to a `novel` element in another document, but since they have different names and (perhaps) different positions in the nesting structure, it is impossible to write a simple XQuery that gets information from both. Therefore, XML subtrees must first be converted to RDF before information about object identity can be inferred.

As previously implied, RDF Schema is not capable of solving the object identity problem. RDF Schema is simply not expressive enough to write rules that allow inferences to be made about object identity. Rather, this is an issue that can best be handled by the OWL, as part of another, higher layer of the semantic integration framework. Amann et al. and other authors refer to object identity in terms of *semantic keys* (1).

Semantic integration allows XML documents to be queried through an ontology. Being able do this allows users to request and manipulate information at a higher level of abstraction. In other words, semantic integration allows the user to sidestep problems like synonymy and polysemy, while taking advantage of a model that understands hypernymy and hyponymy. This technique also enables data that appears in one XML document to be used seamlessly with data from another XML document. This is ordinarily difficult to do at the XML level, where tools and schemas do not have any notion of foreign keys between documents.

CHAPTER 4

DESCRIPTION OF THE FRAMEWORK

4.1 <u>Overview</u>

The semantic integration framework defines a system that uses a layered architecture and consists of the following components:

<u>Global RDF Mediator</u>: The global RDF mediator sits at the top of the system and provides a uniform interface to the user. It tries to emulate an RDF repository with an RDQL query engine (even though the underlying information is stored as XML, not RDF). The global mediator has an associated global ontology (written in RDFS) which defines the classes and properties whose instances populate the repository. The mediator accepts RDQL queries and translates them into XQuery expressions which are forwarded to the local XML repositories. The query results are returned in the table format expected by RDQL. However, the global mediator can operate in two different modes, which determine the nature of the URIs in the table. The implications of these two modes will be discussed in the section that describes the global mediator.

<u>Local XML Repositories</u>: The local XML repositories provide interfaces to the local XML data sources. A given repository may be attached to multiple data sources, but it has only one schema, and each of its data sources must conform to that schema. An XML repository simulates a single, monolithic XML document (even though the real data may be distributed at several sites across the Web). This layer helps to simplify the process of translating RDQL to XQuery, as well as to simplify the process of adding new XML data sources. An XML

repository accepts XQuery expressions and forwards them to each XML data source. It collects the results from each local source and then sends the aggregated results back to the global RDF mediator.

Mapping Structure: The mapping structure is a table, each row of which specifies a mapping. The table is split up into two sections: Classes and Properties. The Classes section contains mappings between RDF classes and XPath expressions; the Properties section contains mappings between RDF properties and XPath expressions. This mapping structure provides the bridge between the global RDF mediator and the local XML repositories. Recall that each XML repository has an associated XML schema. The mapping structure specifies how the elements and attributes defined in the local XML schema map to classes and properties in the global RDF schema (and vice versa). The mapping structure is used for both query translation and result transformation. Mapping structures are stored at the XML repository layer.

Local XML Data Sources: The local XML data sources are the real sources of information. Each local XML data source is simply an XML document or front-end for another type of system that simulates an XML document. A given data source must be associated with one (and only one) XML repository. From the perspective of the integration system, a local data source is valid if and only if it conforms to its repository's schema. In addition, it is desirable for a local XML data source to references its schema directly (perhaps by using the `schemaLocation` attribute from the XML Schema instance namespace).

In the following sections, we will explore the aspects of the semantic integration framework in much closer detail. Figure 7 shows a graphical representation of the framework.
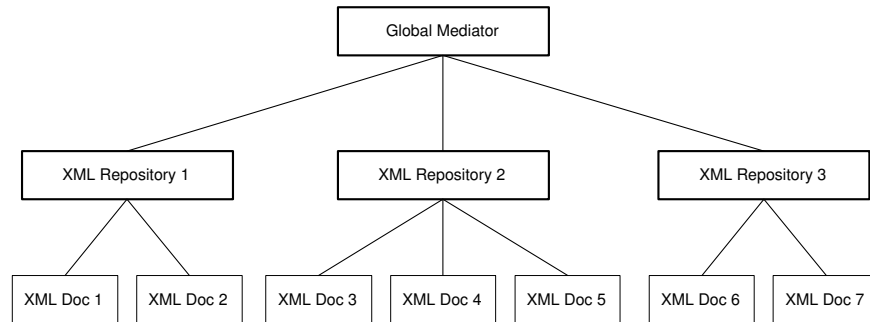
Figure 7: The Semantic Integration Framework

4.2   The Global RDF Mediator

The global RDF mediator provides a front-end for the integration system. It should do so in such a way that the user of the system should not know that the underlying data is not RDF. Although the system can function fairly well in this respect, there are certain differences between the integration system and a real RDF repository. These differences are imposed by the fact that the underlying data is represented as XML. In this section we describe these differences as well as the general characteristics of the global mediator.

4.2.1   Using XPointer for the URIs

As previously stated, the global RDF mediator acts similarly to an RDF repository with an RDQL query engine. In addition, the RDQL engine must be able to return a query result in the proper format—a table whose entries are RDF resources. This means that an RDQL query such as the one in Figure 8(a) should result in a table like the one shown in Figure 8(b). Note that all the URIs are actually URLs, as is typical for an RDF model.

```
SELECT ?artist, ?work
WHERE  (?artist, <art:creates>, ?work)
```

(a) An RDQL query

Artist	Work
⟨http://art.org/directors#Kurosawa⟩	⟨http://art.org/paintings#aZfx90⟩
⟨http://art.org/painters#Caravaggio⟩	⟨http://art.org/paintings#bWdx9⟩
⟨http://art.org/sculptors#Rodin⟩	⟨http://art.org/paintings#fz9Thb⟩
⟨http://art.org/composers#Beethoven⟩	⟨http://art.org/paintings#l3YufJ⟩

(b) RDQL result table

Artist	Work
⟨http://art.org/directors.xml#/directors/director[3]⟩	—
⟨http://art.org/painters.xml#/painters/painter[2]⟩	—
⟨http://art.org/sculptors.xml#/sculptors/sculptor[5]⟩	—
⟨http://art.org/composers.xml#/composers/composer[4]⟩	—

(c) RDQL result table with XPointer links

Figure 8: A simple RDQL query (a) and the RDQL result table for it (b). The RDQL result table in our framework (c)

In the integration system, all of the underlying data is stored in XML documents, so we will want to return the locations of XML elements rather than the locations of RDF resources. This means, however, that the URIs cannot take the usual form, because URLs are not expressive enough to reference a specific element[1] in an XML document. And unlike an RDF resource, an XML element does not have a unique identifier associated with it. Therefore we cannot simply put the ID of a resource after the "#" symbol, as we usually do for RDF.

Therefore we must use another mechanism to "point" to a specific element. In our framework, the URIs all take the form of XPointer links, where the value that comes before the "#"

---

[1]When we talk about elements, we really mean elements and attributes. But our comments about XML elements applies to XML attributes as well.

symbol is the URL of the XML document and the value that comes after the "#" symbol is an XPath expression for a specific node in the document. Figure 8(c) illustrates the result table containing XPointer links. The XPath expression used in the XPointer must be unambiguous, that is, it must match one, and only one, element in the XML document. Also, it must not contain any predicate filters except the position filter[1]. We call such an XPath expression a *normalized XPath expression.*

4.2.2   XPointer Mode

Using XPointer is a good solution for returning results that reflect the real locations of the data[2]. It is also perfectly legal from the perspective of RDF, since URIs do not have to be URLs. Providing the results using XPointer allows a user to go to the data directly, and perhaps even retrieve the DOM subtree for the element pointed to by the XPointer. When the global RDF mediator returns XPointers for its URIs, it is said to be operating in *XPointer mode.* As implied, there is another mode of operation which we will describe in the next section.

4.2.3   Conversion Mode

XPointer mode is useful, but can be cumbersome in some situations. Since only URIs are returned, the XML data has not actually been retrieved, unless the original RDQL query asked for literals to be returned. This means that the user cannot immediately begin to process the data. This can be costly in terms of performance, and is especially undesirable in circumstances where communication costs are high. Also, XPointer mode does not allow for inferences about

---

[1]The XPointers in 4.2.1(c) use abbreviated position filters. For example, the XPath expression `/sculptors/sculptor[5]` could also be expressed as `/sculptor/sculptor[position()=5]`.

[2]The XPointer's URL section (the part that comes before the "#" symbol) actually points to a local XML data source; it does not point to the location of an XML repository.

object identity to be made, since the data remains in XML form, which has no semantics. For this reason the RDF mediator can also return an RDF submodel that is generated from the query results. This is accomplished in the following manner:

1. The RDF global mediator receives an RDQL query, which is translated into XQuery expressions.

2. The XQuery expressions are sent to the XML repositories, which return DOM trees that contain the data from the results.

3. These DOM trees are converted to RDF triples. These triples form a part of the larger RDF model, and are stored in a temporary *result repository* in main memory.

4. The standard result of an RDQL query (table of resources) is also returned. However, URIs in the result table point to resources in the result repository, and thus are only valid while the result repository exists in main memory.

When the global mediator returns an RDF submodel along with the result table, it is said to be operating in *Conversion mode*. True to its name, Conversion mode converts the information stored in DOM trees into RDF triples. If the DOM trees returned by the XML repositories are not expected to be large, then Conversion mode can have a performance advantage over XPointer mode. This is because all of the relevant data is returned at once and is available for immediate processing (by RDF tools). Also, since the data is now available as RDF, inferencing can be used by a rule engine that understands RDF. Inferencing allows new information to be "discovered"; in particular, statements about object identity can be added to the model. This allows information from different documents to be shared and combined in meaningful ways.

Conversion mode differs from XPointer mode in two major ways: (1) the format of the URIs; and (3) the amount and type of data returned. Once the results have been processed, the contents of the result repository can be discarded. The result repository can be thought of as a cache for query results.

4.3   The Local XML Respositories

A local XML repository provides a facade for the underlying XML data by giving the illusion of being a single, monolithic XML document. It also provides distributed query processing for multiple XML documents. Using an XML repository simplifies the process of translating RDQL to XQuery, because single-source RDQL queries do not have to be translated to multiple-source XQueries. Also, an XML repository layer makes it easier to add new XML data sources to the integration system. An XML repository can add a new local data source without notifying the global mediator because the global mediator only deals with the XML repository. This decoupling makes the integration system easier to manage.

4.3.1   The Role of XML Schema

The XML repository integrates multiple XML documents, all of which conform to the same schema. This schema, encoded in W3C Schema or some other schema language, must be stored at the XML repository. The XML repositorys schema is used to validate local XML data sources that are added to the integration system. If an XML document does not conform to a known schema, then the following procedure must be performed: (1) Find the schema for the new XML document (if none exists, then abort); (2) Create a new XML repository that owns that schema; (3) Add the XML document to the newly-created XML repository.

4.3.2 Distributed Query Processing

The XML repository must handle distributed query processing for its local data sources. There two main approaches: The naive approach is to union all of the local data sources into a single document and then query that unified document; this is simple to implement but very costly. The more efficient approach is to propagate the XQuery to each local data source and then aggregate the results. The XQueries that the integration system deals with are relatively simple because they are transformed from RDQL. The XQuery results are also relatively simple; they are DOM trees which can easily be manipulated by any DOM level 2 API. Our focus is not on the optimization of distributed XQuery processing, so we will not discuss it further. However, Suciu has a written extensively about this topic (21).

4.4 The Mapping Structure

The mapping structure helps the semantic integration system bridge the gap between XML and RDF. This gap is a fairly large one, as these two languages serve very different purposes. XML is a semistructured language capable of representing many different kinds of data. On the other hand, RDF is a language that can encode data and explicitly describe the semantics of the data. The schema languages for XML and RDF are also very different. XML schema languages describe the labels and structure of XML documents but not the inherent meaning of the documents. RDF Schema describes a simple ontology that models the semantic knowledge contained in RDF models.

The mapping structure is stored at the XML respository layer. It is a synthesis of conceptual information (from the global ontology) and structural information (from the local XML schema). Therefore, a mapping structure is specific to the XML repository it resides at. The mapping

structure has two main purposes: (1) Enable RDQL to be translated to XQuery; (2) Enable DOM trees to be transformed into RDF triples. These two purposes are closely-related.

4.4.1 Triples to XPath

It must be noted that the mapping structure does not, by itself, perform any actions. Instead, it provides a mapping between RDF triples and XPath expressions. This allows query translation because of the following rationale:

1. An RDQL query specifies a submodel that should be found within a model.

2. An RDF model is an unordered collection of triples.

3. Using facts 1 and 2, we can say that an RDQL query can be modeled as a collection of triples.

4. A simple XQuery[1] can be represented as a subtree.

5. An XPath expression describes a single path through a tree. A collection of XPath expressions can be merged to form a subtree[2].

6. Using facts 4 and 5, we can say that a simple XQuery can be modeled as a collection of XPath expressions.

7. Using facts 3 and 6, we can say that RDQL query translation is equivalent to mapping a collection of RDF triples to a collection of XPath expressions.

---

[1]Later we will define more precisely what we mean by a "simple XQuery". For now, we can think of it as a non-nested XQuery that be modeled as a tree data structure.

[2]For example, merging url/parent/child1 and `/parent/child2` gives a subtree with parent as the root, and child1 and child2 as child nodes of parent.

The above reasoning simplifies the issue for the sake of explication, but gives the general idea of why the mapping structure provides sufficient support for query translation. Although result transformation, or the conversion of XML to RDF, is a different problem, the mapping structure can still be used. This is because we can use similar justification: A DOM tree is a collection of XPath expressions, an RDF model is a collection of triples, and so mapping XPath expressions to triples will allow us to convert a DOM tree to an RDF model.

Although the mapping structure is not much more than a table containing mappings between triples and path expressions, it is divided into two sections: the Classes section and the Properties section. The main reasons for this division are convenience and performance.

4.4.2 Classes Section

The Classes section of the mapping structure contains information about what RDF classes appear in a particular XML schema. Although only the name of the class is shown in the mapping sturcture, it is really a triple of the form (Resource type Class). Since all entries in the Classes section have the same structure, we omit the "Resource type" part from the triple. The classes section can only map "type triples"—triples that use the `rdf:type` property in the predicate slot. To find a mapping for a type triple of the form (A type B), we must search for subclasses of class B. For example, to map (?person type Person), we must search the Classes section for subclasses of `Person`. Any subclass of `Person` is a valid match, so if `Student` is found in the Classes section, it will be used for the mapping. Note that the subclass property is inclusive (meaning `Person` is also a subclass of `Person`), so that `Person` will be match mappings for `Person`.

4.4.3   Properties Section

The Properties section of the mapping structure contains information about what RDF properties appear in a given XML schema. These triples have the form (Class Property Class). The process of finding a valid mapping in the Properties section is a bit more complicated than for the Classes section. The idea is to search for *sub-triples*, or specialized triples. A triple (a, b, c) is a sub-triple of (x, y, z) if and only if: a is a subclass of x, b is a subproperty of y, and c is a subclass of z. So to map the triple (?student studies ?major), we have to find a mapping that contains a sub-triple of (Student studies Major). For example, a triple such as (GradStudent studies ComputerScience) would be a valid match.

4.4.4   Characteristics of the Mapping Structure

An example mapping structure is shown in Figure 9. It is essentially a table where the left-hand column contains RDF triples, and the right-hand column contains XPath expressions that the triples are mapped to. Besides the two different columns, the table is also divided into sections: The Classes section and the Properties section. The Classes section allows mappings to be made based on a subclass search. The Properties section allows mappings to be made based on a sub-triple search (sub-triples were defined in the previous section). To get the most possible matches, the triples in the Classes and Properties sections must be as specific as possible. This means that, if an XML repository contains information about `Painters`, then its mapping structure should not have (Resource, type, Person) in its Classes section. The triple (Resource, type, Painter) should be used instead because it supercedes (Resource, type, Person), since `Painters` is a subclass of `Person`. Thus, queries about `Painter`s and `Person`s can both be translated using this mapping structure.

Classes	
Painter (Resource, type, Painter)	painters/painter
Painting (Resource, type, Painting)	painters/painter/painting
Properties	
(Painter, paints, Painting)	painters/painter/painting
(Painter, name, Literal)	painters/painter/@name
(Painting, title, Literal)	painters/painter/painting/@title

Figure 9: A mapping structure

In our framework, not all possible XPath expressions will be found in the mapping structure. This is because the triples in the mapping structure come from the global ontology, and the global ontology may not have an equivalent class or property for every XML element. Recall that the global ontology provides allows the integration system to provide a view to the underlying local data. It is up to the designer of the global ontology to decide what kind of information will be available through this view.

Currently, our framework has no provision for semi-automatically generating the mapping structure. Currently the mapping structure must be produced by hand, so it is not difficult to do so. However this is a serious deficiency for large-scale integration and so semi-automatic generation will be a topic for future work. Camillo et al have done some work in this area (5).

One optimization issue that should be considered is the question of answerability. Some XML repositories in the integrated system will not contain the information needed to answer a particular query. For instance, if an RDQL query references the `Painting` class and the `paints` property, then an XML repository containing information about sculptors will not be

relevant to the query. If an RDQL's triples cannot match any of the triples in a particular XML repository's mapping structure, then query translation will not be initiated for that repository.

4.5 Local XML Data Sources

A local XML data source is a single XML document, or an entity that acts like an XML document. For example, a relational database that has an XML interface could be considered a local XML data source. It is best if a local XML data source references its own schema. This can be done in the case of W3C schema by using the `schemaLocation` attribute found in the XML schema instance namespace:

```
<example xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://example.org/example.xsd">
```

If a local XML data source references its own schema, it is easy to easy to determine whether it conforms to a known XML schema. If it does, it will be validated by the appropriate XML repository and added if it is determined to be valid. If it is unknown, a new XML repository must be created which uses that schema. An XML document that does not conform to a known schema cannot be added to the integration system by itself; it must be accompanied by its schema. This is necessary because if the document's schema is unknown, that schema must be made available or the new XML repository cannot be created.

4.6 Query Translation

Query translation in the semantic integration system involves translating a single-source RDQL query to a single-source XQuery. Basically, query translation makes use of the mapping structure, and "rephrases" the query in one language to another, somewhat similar language. Recall that RDQL has five clauses: `SELECT`, `FROM`, `WHERE`, `AND`, and `USING`. Similarly, XQuery

RDQL	Equivalent structure(s) in XQuery
SELECT	Very similar to the RETURN clause.
FROM	The RDQL FROM clause is not used by the integration system because all queries are executed against the global ontology view. However, the FROM clause roughly corresponds with the doc/document function in XQuery.
WHERE	The RDQL WHERE clause can be approximated with LET and FOR clauses.
AND	The RDQL AND clause is very similar to the XQuery WHERE clause.
USING	No equivalent clause, but similar to the declare namspace statement. However the USING does not need to be translated because it a language construct used only for readability.

TABLE I

RDQL CLAUSES AND THEIR EQUIVALENT STRUCTURE(S) IN XQUERY

uses the FLWR syntax and thus has four clauses: `for`, `let`, `where`, and `return`. Table I provides a brief overview of how the clauses match up.

The most difficult aspect of the query translation process is translating the RDQL `WHERE` clause to a sequence of XQuery `for` clauses. The basic technique that we employ is to take the triples from the `WHERE` clause and map them to XPath expressions (using the mapping structure). After that we merge the XPath expressions to form an *XQuery tree*. An XQuery tree is a data structure that can easily be converted to a series of `for` clauses. The rationale for

this approach has already been covered in the 4.4.1. We now give the procedure for translating
the RDQL `WHERE` clause to a sequence of XQuery `for` clauses:

1. Parse RDQL query.

2. Extract triples from `WHERE` clause.

3. Expand abbreviated URIs.

4. Do type resolution on triples.

5. Map triples to XPath expressions.

6. Merge XPath expressions to form XQuery tree.

7. Convert XQuery tree into an XQuery expression.

 The detailed explanation of each step in the above procedure is shown below:

1. *Parse the RDQL query.*

   In this step, we simply break the RDQL query into its constuent clauses (`SELECT`, `WHERE`,
   `AND`, `USING`).

2. *Extract triples from `WHERE` clause.*

   The `WHERE` clause of an RDQL query is always a list of unordered triples. They are very
   easy to extract because they all have the form (subject, predicate, object).

3. *Expand abbreviated URIs.*

   We use the list of mappings found in the `USING` clause to expand the URIs in the triples.

4. *Do type resolution on triples.*

   For variables that occur in the subject and object slots of the triple, we can use the domain

and range of the predicate to infer the type. If the predicate does not have a domain and range, the default type of a variable is Resource (since every other class derives from Resource). There is no way to infer the type of predicate variables, so we set them to the default type of Property.

5. *Map triples to XPath expressions.*

   After type resolution, we try to map each triple of the WHERE clause to a corresponding XPath expression in the mapping structure. However, type triples are treated differently from other triples:

   - Type triples, or those triples which use rdf:type as the predicate, are mapped using the Classes section. A simple subclass search is used to find the appropriate mapping.

   - All other triples are mapped using the Properties section. A sub-triple search is used to find the appropriate mapping. In other words, a triple t is mapped if a sub-triple of t can be found in the left-hand column of the mapping structure.

6. *Merge XPath expressions to form XQuery tree.*

   Once all the triples have been mapped to XPath expressions, we merge all XPath expressions to form a tree structure called the XQuery tree.

7. *Convert XQuery tree to XQuery expression.*

   We use depth-first traversal on the XQuery tree to obtain a single let clause followed by a sequential list of for clauses. This forms the main part of the translated XQuery expression.

Besides the translation of the RDQL `WHERE` clause to XQuery LET and FOR clauses, we must also:

1. Translate the RDQL `SELECT` clause into the XQuery `return` clause.

2. Translate the RDQL `AND` clause into the XQuery `where` clause.

Translating the `SELECT` clause into the `return` clause is easy because they are almost identical. RDQL and XQuery have different boolean operators, but string substitutions can be used to deal with that issue. Likewise, translating the `SELECT` clause to the `return` clause is fairly straightforward, since all that must be done is to translate the RDQL variable names to the XQuery variable names. The information from the original RDQL clauses is encoded in the XQuery tree and used to generate the full XQuery expression. The XQuery tree is not a very complex structure, and so does not produce very complex XQueries. We call these queries *tree-based XQuery expressions.*

Query translation is not very different between XPointer mode and Conversion mode (the main difference between these modes is in how the results are transformed). However, there is one subtle difference: XPointer mode quits if a complete query translation cannot be made; Conversion mode will continue even if only a partial query translation is possible. To understand why, we should review the limitations of XPointer mode.

XPointer mode is meant to retrieve only the locations of the XML nodes that can satisfy a given query. It performs no transformations on the underlying XML data. XQuery is not good for sharing or combining data that comes from documents with different schemas (i.e. XQuery is not good at handling schematic heterogeneity). Therefore an XQuery cannot, in general, be answered with a respect to a specific document unless the document contains all

the information needed to answer the XQuery. When some RDF triples in an RDQL query cannot be mapped, that means the XML repository that owns the mapping structure does not have all of the information needed to answer the RDQL query.

Unlike XPointer mode, Conversion mode always transforms the underlying XML data to RDF triples. When DOM trees are converted to RDF triples, the data loses its structural properties and gains semantic properties (the most important of which is typing information). This makes the transformed RDF data much better for sharing and combining than when it was in its original XML form. In Conversion mode, an RDQL query whose triples cannot all be mapped will still be translated, in the hopes that the information returned from a partially-mapped query will be useful when combined with information from other sources. Therefore, Conversion mode is much better poised to take advantage of situations where there is an overlap of information content between different data sources.

4.7   Result Transformation

The result transformation process is completely different for XPointer and Conversion mode. The major difference rests in what happens to the DOM trees returned by XQuery. In XPointer mode, the DOM trees are used to obtain normalized XPath expressions, and then appended to URLs to make them XPointers. In Conversion mode, the DOM trees are translated to RDF triples with help from the mapping structure.

4.7.1   Result Transformation in XPointer Mode

Result transformation in XPointer mode is fairly straightforward. After the XQuery returns a DOM tree, the child nodes of the DOM tree are examined to determine their normalized XPath

expressions. These XPath expressions are then appended to the document's URL to form an XPointer link. The XPointer links are then used as URIs in the RDQL result table.

4.7.2 Result Transformation in Conversion Mode

Result transformation in Conversion mode is much more complicated than in XPointer mode. The result DOM tree of the XQuery must be converted to an RDF submodel. This RDF submodel can also be thought of as an unordered collection of RDF triples. The mapping structure provides the mapping information needed for the conversion process. However, instead of mapping triples to XPaths, we now map XPath expressions to triples. The end product of the result transformation process is a result repository, containing the transformed RDF triples, and an RDQL result table, which contains URIs of resources in the result repository. Below we present the procedure for result transformation in Conversion mode:

1. In the result DOM tree, get all the level-2 subtrees (subtrees whose roots are the child nodes of the root node of the DOM tree).

2. Search each subtree to see if it contains XPath expressions that can be mapped to RDF triples.

3. For each mapping that is found, convert the given XPath expression to an RDF triple and add it to the result repository. The resources in the triples are assigned semi-random URIs.

4. Since the result respository is an RDF model, we can query it using RDQL. So to complete the process, we run an RDQL query on the result repository to obtain the RDQL result table.

In 4.7.2 we have a diagram of the semantic integration process in XPointer mode. A similar diagram for Conversion mode is given in 4.7.2.
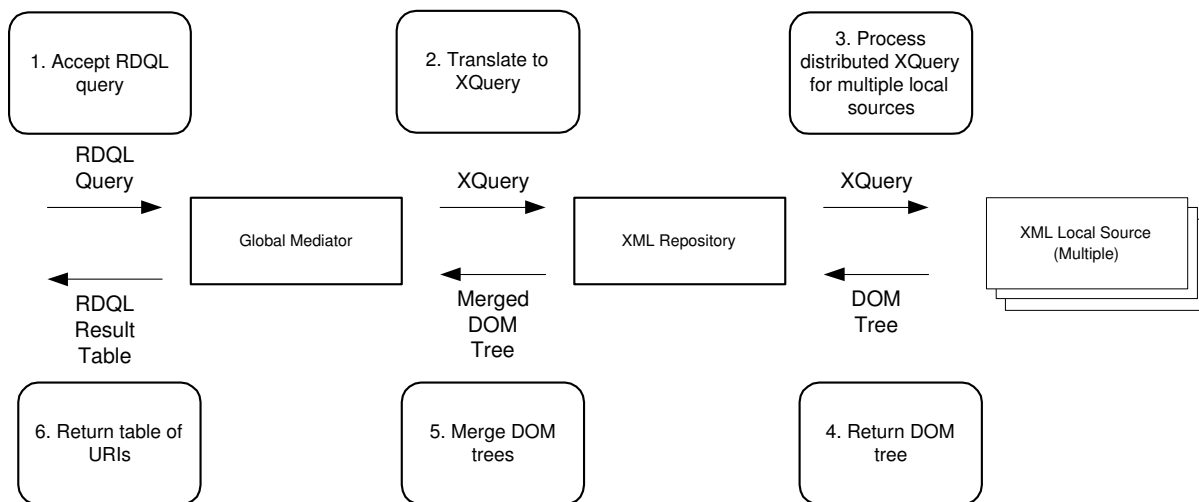

Figure 10: Query translation and result transformation in XPointer mode

4.8   Limitations

There are a number of important limitations that somewhat lessen the usefulness of the semantic integration framework. The first limitation is that there is no provision for specifying individual resources in the RDQL query. For example, we cannot write an RDQL query like this:

```
SELECT ?name
WHERE (<http://example.org/people.xml#people/person[3]>, dc:name, ?name)
```
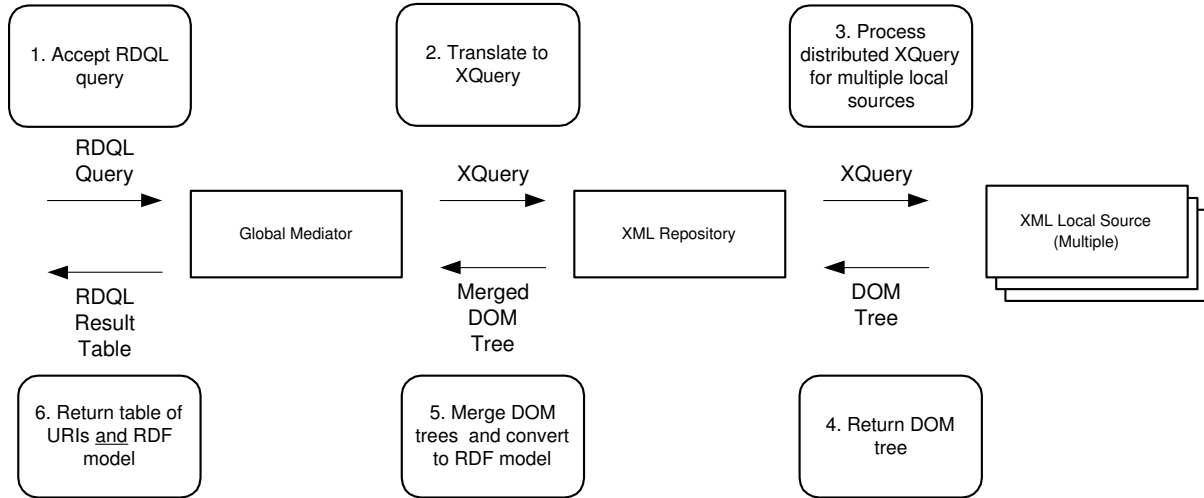
Figure 11: Query translation and result transformation in Conversion mode

It should be possible to reference an individual resource, but as of this writing we have not figured out a mechanism to accomplish this. Second, the mapping structure only accepts absolute XPath expressions in the right-hand column. This means that XML elements that can occur at different levels might not be found in the translated XQuery. This limits the kinds of schemas for which mapping structures can be made. There is a way to specify few alternate XPath expressions to be used, but in general there is no way around this limitation. If the mapping structure were to accept arbitrary XPath expressions on the right-hand side, it might make it impossible to merge the XPath expressions to form the XQuery tree.

CHAPTER 5

IMPLEMENTATION

A research implementation of the semantic integration system has been carried out. It is merely a prototype and is thus incomplete, but it does include support for query translation and result transformation (though only in XPointer mode). Although far from perfect it serves as a basis for further implementation efforts. In the following sections we describe the languages and libraries we used, as well as some relevant details of the implementation.

5.1   Languages

Java is a popular interpreted language developed by Sun Microsystems. Java source code is statically compiled to bytecode, which is executed by a Java virtual machine. Since Java virtual machines exist for every major platform, Java programs can be compiled on one platform and deployed on another. This is in line with Javas slogan of "Write once, run anywhere". One of the most interesting aspects of Java is its Reflection API. Reflection allows a Java program to query a class for its methods, fields, subclasses, etc. In addition, the Reflection API allows Java objects to be manipulated indirectly—for instance, an object can be instantiated even if its class was not known at compile-time. Also, an objects methods can be invoked even if they were not known at compile-time.

The facilities provided by the Reflection API allow a plethora of scripting languages to be built on top of Java. One of the most powerful and useful of these scripting languages is Jython. Jython is an implementation of the Python programming language built on top of Java. Because of Java's Reflection API, Jython can easily access Java classes and their

corresponding methods. Therefore Jython programming is very similar to Java programming, except that the Jython programmer does not have to deal with Java's C-based syntax nor its static typing. Although the libraries we used in the implementation were all Java libraries, we programmed the prototype integration system in Jython because Jython code is more compact and expressive. We were able to do it much more quickly than if we programmed it directly in Java. However, for performance reasons we may consider rewriting some of the modules in Java.

5.2   Libraries

There are many RDF libraries now, but the most complete library is probably Jena from HP Labs. Jena is coded in Java and provides support for serialization, relational database storage, and querying through RDQL. Best of all, it provides an ontology API and comes with rule engines for basic inferencing on RDF Schemas. It also has limited support for both DAML+OIL and the variants of OWL. We use Jena to load the global ontology from file and to query the global ontology for subclass/subproperty relationships.

XQuery is a W3C recommendation and is already a major query language for XML. There are various free and commercial implemenations of XQuery, most of them in Java. IPSI-XQ, from the Fraunhofer Institute for Integrated Publications and Information Systems, is a free XQuery implementation. We chose it because it comes with an excellent GUI interface (for testing XQueries), and decent documentation.

The most popular Java library for XML processing is the Xerces-J API, from the Apache Software Foundation. Both Jena and IPSI-XQ make use of Xerces-J, but we also use it directly to access and manipulate the DOM trees that are returned by XQuery.

5.3   Details

The parsing of the RDQL query was handled using regular expressions. RDQL is a fairly simple language, so this was not too difficult. We did not use an abstract syntax tree to store the query; instead we simply separated the classes and then used more regular expressions to convert the clauses to their respective data structures. For example, the `WHERE` and `SELECT` clauses are converted to a list of triples, where selected elements of the triples were marked. The `AND` clause is converted to a list of boolean expressions. The `USING` clause is converted to a dictionary object where the keys are the prefixes and the values are the namespaces they abbreviate.

Query translation was used Jenas Ontology API to search for subclasses and sub-triples. One obstacle to this was the fact that Jena mostly uses interfaces to represent its ontology classes. Therefore we were unable to take advantage of subclassing, which made the design a little more awkward.

The implementation of result transformation was not as straightforward as we initially thought it would be. The major problem is that there is no easy way to get XQuery to return normalized XPath expressions. Even though XQuery returns a DOM tree, this DOM is not connected to the DOM tree of the XML document. In other words, the DOM tree that XQuery returns is a copy of a subtree of the queried document. Therefore, it is not possible to traverse upwards through the tree nodes to derive the normalized XPath. Because of these problems, we were forced to implement XPointer modes result transformation using the following procedure:

1. Execute the XQuery, and get DOM tree T_Q as the result.

2. Let R be the root node of T_Q, and let T be the original DOM tree (of the queried document).

3. For each child node C_Q of R, find the equivalent node C in T (use depth-first search for node comparison).

4. For each node C, traverse upwards through T to derive the normalized XPath. This is trivial, just use the getParentNode() and getElementsByTagName() methods.

The solution delineated above is correct but inelegant because it forces the XML document to be parsed twice: once by the query, the second time by the DOM parser. There are two potential ways to solve this problem: modify the source of the XQuery engine to keep track of the current path during query execution, or write an XQuery function to return the normalized path. The first suggestion is possible, but perhaps not very feasible because of the amount of time and effort it would require. The second suggestion holds some promise, because XQuery is designed to be a functional programming language, rather than just a query language. However, the IPSI-XQ implementation does not seem to allow the relative position of a node to be returned. More specifically, the position() function does not seem to work outside of predicate filters. Perhaps more mature implementations of XQuery will allow us to optimize the result transformation under XPointer mode. Note that Conversion mode is unaffected by these issues, because Conversion mode is not concerned with where the DOM nodes came from, but rather with how to convert DOM nodes into RDF triples.

Unfortunately, as of this writing Conversion mode has not been implemented. This means that the DOM trees from the XQuery expressions cannot be converted to RDF triples. This is the most important omission from the prototype and should be rectified in future versions.

Also missing from the integration system is distributed query processing. Thus, instead of using XML repositories, we simply use single XML documents. From the perspective of the global mediator there is little difference because an XML repository is supposed to simulate a large, monolithic XML document. Thus we do not feel this is a big detraction, because distributed XQuery processing is not the focus of this project.

CHAPTER 6

EXAMPLES

Here we present a detailed example that illustrates, in a concrete manner, the ideal behavior of the semantic integration system.

We will present a two XML repositories whose schemas are expressed in the Relax NG schema language. We choose Relax NG because its compact form is very easy to read and has the same cardinality operators as the well-known DTD schema language. It has a level of expressiveness on par with W3C XML Schema. But just because we use RelaxNG does not mean that our approach does not apply to similar schema languages like W3C XML Schema. In the following examples we have decided to abstract away the XML repository and just use single XML documents. This serves to simplify the examples but does not sacrifice the larger issues since the XML repository is meant to simulate a single XML document.

For the global ontology we will use Notation 3, an RDF language which is more compact and readable than the XML syntax commonly used on the web.

6.1 Authors and Books

This first example is a very basic example of semantic integration in XPointer mode. The example comes from the publishing domain, and it deals with information concerning books and their authors.

File Name	Type
authors_books.n3	Global Ontology
authors.ng	XML Schema
books.ng	XML Schema
authors.xml	XML Repository
books.xml	XML Repository
authors.map	Mapping Structure
books.map	Mapping Structure

TABLE II

RELEVANT FILES FOR THE AUTHOR AND BOOKS EXAMPLE

6.2   Overview of the Examples

For the sake of clarity, we will keep this example as simple as possible. There are seven files relevant to the example, show in Table II. The two sample queries are run against these seven files. Both sample queries are running in XPointer mode.

6.3   First Query and Results

The first RDQL query is shown in Figure 16. 6.3 shows the RDQL result table that is produced. The intervening figures show the various stages of processing that are needed to obtain these results.

6.4   Second Query and Results

The second RDQL query is shown in 6.4. 6.4 shows the RDQL result table that is produced. The intervening figures show the various stages of processing that are needed to obtain these results.
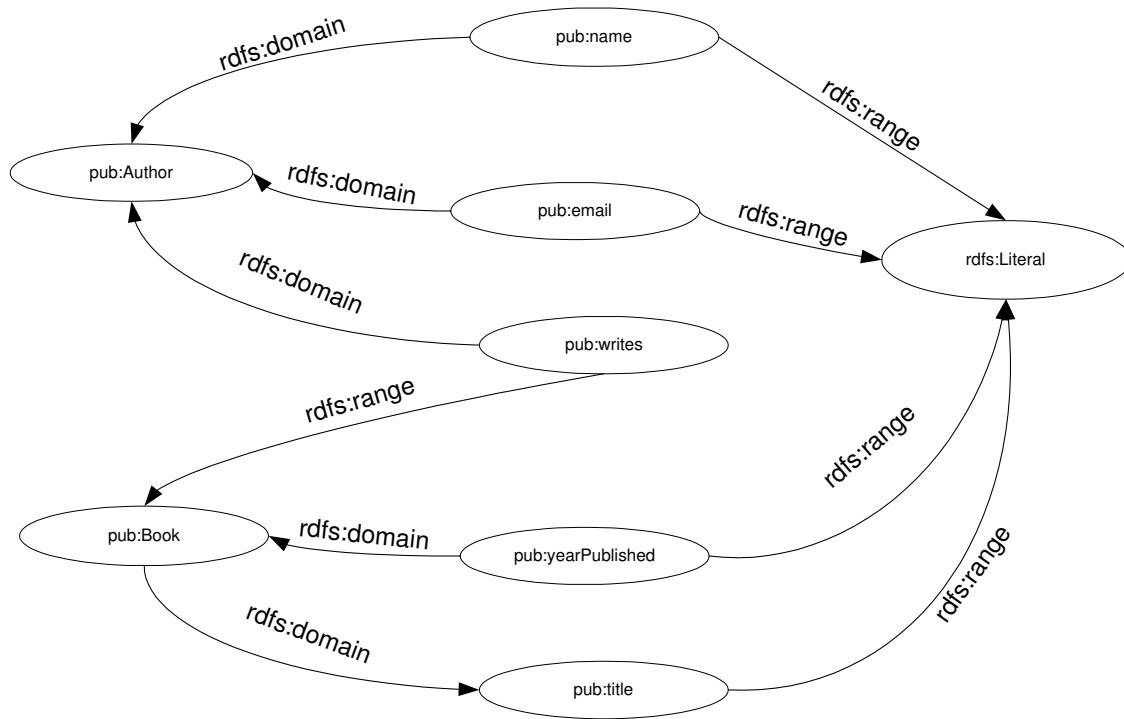
Figure 12: Publishing Ontology

```
element authors {
  element author {
    attribute name { text },
    attribute email { text },
    element book {
      attribute title { text }
    }*
  }+
}
```

(a) authors.ng

```
element books {
  element book {
    attribute title { text },
    attribute yearPublished { text },
    element author {
      attribute name { text }
    }*
  }+
}
```

(b) books.ng

Figure 13: XML schemas in Relax NG

```
<authors>
<author name="Huiyong Xiao"
        email="hxiao@uic.edu">
  <book title="Semantic Integration"/>
  <book title="Java Squared"/>
</author>
<author name="Flora Wu"
        email="fwu@pyblast.org">
  <book title="Python Unlimited"/>
  <book title="Java Squared"/>
</author>
<author name="Feihong Hsu"
        email="fhsu@dork.net">
  <book title="Jython in 24 Hours"/>
</author>
</authors>
```

(a) authors.ng

```
<books>
<book title="Semantic Integration"
      yearPublished="2004">
  <author name="Huiyong Xiao"/>
</book>
<book title="Python Unlimited"
      yearPublished="2005">
  <author name="Flora Wu"/>
</book>
<book title="Java Squared"
      yearPublished="2005">
  <author name="Flora Wu"/>
  <author name="Huiyong Xiao"/>
</book>
<book title="XML for Pros"
      yearPublished="2006">
  <author name="Olga Sayenko"/>
</book>
</books>
```

(b) books.ng

Figure 14: XML repositories

Classes	
pub:Author pub:Book	authors/author authors/author/book
Properties	
(pub:Author, pub:name, rdfs:Literal) (pub:Author, pub:email, rdfs:Literal) (pub:Book, pub:title, rdfs:Literal) (pub:Author, pub:writes, pub:Book)	<u>authors</u>/author/<u>@name</u> <u>authors</u>/author/<u>@email</u> authors/author/<u>book</u>/<u>@title</u> authors/<u>author</u>/<u>book</u>

(a) authors.ng

Classes	
pub:Author pub:Book	books/book/author books/book
Properties	
(pub:Book, pub:title, rdfs:Literal) (pub:Book, pub:yearPublished, rdfs:Literal) (pub:Author, pub:name, rdfs:Literal) (pub:Author, pub:writes, pub:Book)	books/<u>book</u>/<u>@title</u> <u>books</u>/book/@yearPublished books/book/<u>author</u>/<u>@name</u> books/<u>book</u>/<u>author</u>

(b) books.ng

Figure 15: XML repositories

```
SELECT ?book, ?author
WHERE (?author, <pub:writes>, ?book)
USING pub for NAMESPACE <http://example.org/publishing#>
```

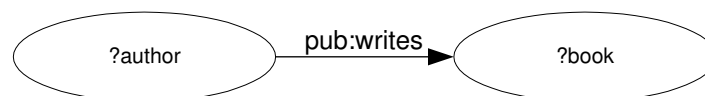Figure 16: An RDQL query that returns a listing of authors and their books

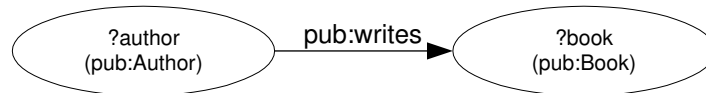Figure 17: The WHERE clause of the RDQL query, in graph form

Figure 18: The WHERE graph after type resolution

Triple	Type Triple	Mapped XPath Expression
(?author, pub:writes, ?book)	(pub:Author, pub:writes, pub:Book)	authors/<u>author</u>/<u>book</u>

(a) Mappings from authors.map

Triple	Type Triple	Mapped XPath Expression
(?author, pub:writes, ?book)	(?pub:Author, pub:writes, pub:Book)	books/<u>book</u>/<u>author</u>

(b) Mappings from books.map

Figure 19: Mappings that result from searching of mapping structures
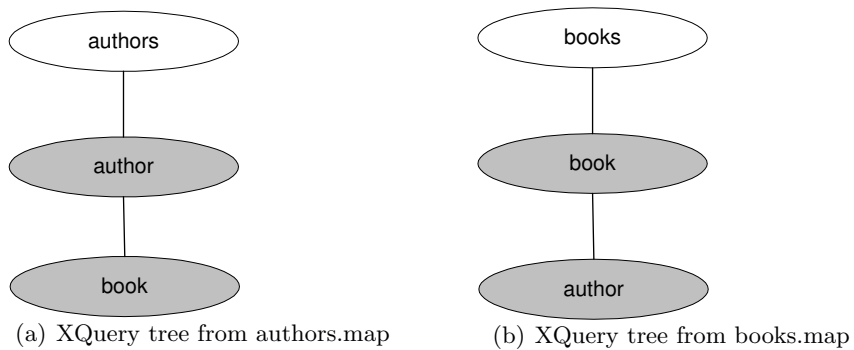
(a) XQuery tree from authors.map

(b) XQuery tree from books.map

Figure 20: XQuery trees generated from successful mappings

```
let $authors := doc("authors.xml")/authors        let $books := doc("books.xml")/books
for $author in $authors/author                     for $book in $books/book
  for $book in $author/book                          for $author in $book/author
  return ($book, $author)                            return ($book, $author)
```

(a) XQuery for authors.xml   (b) XQuery for books.xml

Figure 21: XQuery expressions obtained from XQuery trees

book	author
doc(authors.xml)/authors/author[1]/book[1]	doc(authors.xml)/authors/author[1]
doc(authors.xml)/authors/author[1]/book[2]	doc(authors.xml)/authors/author[1]
doc(authors.xml)/authors/author[2]/book[1]	doc(authors.xml)/authors/author[2]
doc(authors.xml)/authors/author[2]/book[2]	doc(authors.xml)/authors/author[2]
doc(authors.xml)/authors/author[3]/book[1]	doc(authors.xml)/authors/author[3]
doc(books.xml)/books/book[1]	doc(books.xml)/books/book[1]/author[1]
doc(books.xml)/books/book[2]	doc(books.xml)/books/book[2]/author[1]
doc(books.xml)/books/book[3]	doc(books.xml)/books/book[3]/author[1]
doc(books.xml)/books/book[3]	doc(books.xml)/books/book[3]/author[2]
doc(books.xml)/books/book[4]	doc(books.xml)/books/book[4]/author[1]

Figure 22: RDQL results table

```
SELECT ?title, ?name
WHERE (?author, <book:writes>, ?book),
      (?author, <book:name>, ?name),
      (?book, <book:title>, ?title)
USING pub for NAMESPACE <http://example.org/publishing#>
```

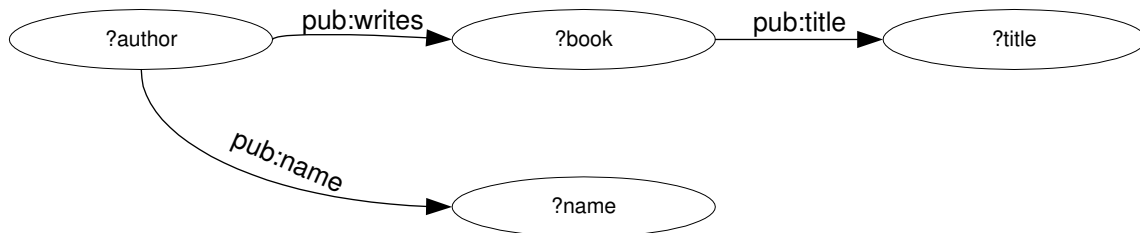Figure 23: An RDQL query that asks for a listing of book titles and their associated authors'
names

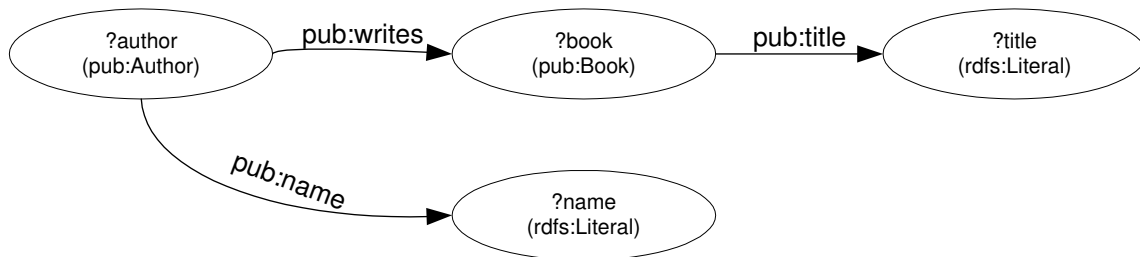Figure 24: The WHERE clause of the RDQL query, in graph form

Figure 25: The WHERE graph after type resolution

Triple	Type Triple	Mapped XPath Expression
(?author, pub:writes, ?book)	(pub:Author, pub:writes, pub:Book)	authors/author/book
(?author, pub:name, ?name)	(pub:Author, pub:name, rdfs:Literal)	authors/author/@name
(?book, pub:title, ?title)	(pub:Book, pub:title, rdfs:Literal)	authors/author/book/@title

(a) Mappings from authors.map

Triple	Type Triple	Mapped XPath Expression
(?author, pub:writes, ?book)	(pub:Author, pub:writes, pub:Book)	books/book/author
(?author, pub:name, ?name)	(pub:Author, pub:name, rdfs:Literal)	books/book/author/@name
(?book, pub:title, ?title)	(pub:Book, pub:title, rdfs:Literal)	books/book/@title

(b) Mappings from books.map

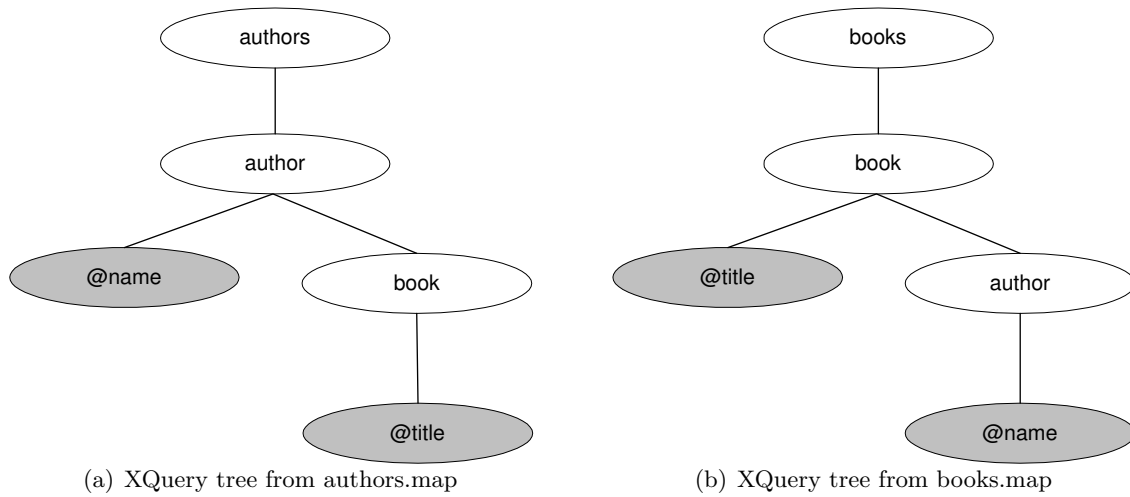Figure 26: Mappings that result from searching the mapping structures

(a) XQuery tree from authors.map

(b) XQuery tree from books.map

Figure 27: XQuery trees generated from successful mappings

title	name
Semantic Integration	Huiyong Xiao
Java Squared	Huiyong Xiao
Python Unlimited	Flora Wu
Java Squared	Flora Wu
Jython in 24 Hours	Feihong Hsu
Semantic Integration	Huiyong Xiao
Python Unlimited	Flora Wu
Java Squared	Flora Wu
Java Squared	Huiyong Xiao
XML for Pros	Olga Sayenko

Figure 28: RDQL results table

CHAPTER 7

CONCLUSION

XML is worthy of becoming a core language of the Web, but its very flexibility makes integration of XML documents very difficult. Instead of trying to integrate at the XML level, we can instead use semantic integration. Semantic integration solves the problem of schema-dependent queries, but it also allows the user to query at a higher level of abstraction. With basic support for inferencing, it becomes possible to use simpler queries with implicit semantic information rather than complex queries with explicit structural information. Semantic integration using RDF schemas and the RDQL query language is ideal because it leverages existing standards and technologies, and also allows us to organize our integration framework around distinct layers. In fact, even higher levels of abstraction can achieved by adding another layer on top of our framework, such as an OWL layer that fits on top of the RDF layer.

7.1 Future Work

Our approach solves certain problems, but there still many problems that need to be addressed. Chief among them is the problem of object identity—for instance, how does one know that the artist named "Van Gogh" in document X represents the same artist named "Van Gogh" in document Y? When we extract the object from the documents we might decide that two `Artist` instances with the same name are in fact the same object. But in real life one does not just use one's name as formal identification. Rather, most people use their social security numbers or driver's license numbers to identify themselves to financial institutions. Sometimes, a combination of properties may be needed to uniquely identify an object, similar to candidate

64

keys in relational databases (Amann et al. calls these semantic keys (1)). In our opinion, the issue of object identity is the biggest obstacle in the path to true semantic integration, and RDF by itself is not be very good at solving it. Therefore future efforts must be directed towards a higher layer to the semantic integration framework which can understand OWL.

Conflict resolution is another difficult issue—what if two XML documents disagree? That is, what if a contradiction arises in the model? Which source is to be trusted? Are both sources suspect? What if the disagreement is very subtle—for example, a word that appears to be different because it is rendered in a foreign language, even though the semantics of the word is the same in either language? These issues also need to be addressed, either through higher-level layers or by sandwiching additional layers in the middle of the framework.

CITED LITERATURE

1. Amann, B., Beeri, C., Fundulaki, I., and Scholl, M.: Ontology-Based Integration of XML Web Resources. In Proceedings of the 1st International Semantic Web Conference (ISWC 2002), pages 117C-131, 2002.

2. Clark, J. and DeRose, S.: XML Path Language (XPath) 2.0. `http://www.w3.org/TR/xpath20/`, W3C Working Drfat, November 2003.

3. Boag, S., Chamberlin, D., Fernández, M. F., Florescu, J. R. D., and Siméon, J.: XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery`, W3C Working Draft, August 2003.

4. Guha, D. and Brickley, R.: RDF Vocabulary Description Language 1.0: RDF Schema. `http://www.w3.org/TR/rdf-schema`, W3C Working Draft, January 2003.

5. Camillo, S. D., Heuser, C. A., and Mello, R. S.: Querying Heterogeneous XML Sources through a Conceptual Schema. In Proceedings of the 22nd International Conference on Conceptual Modeling (ER2003), pages 186C-199, 2003.

6. Chen, Y. and Revesz, P.: CXQuery: A Novel XML Query Language. In Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Science, and Medicine on the Internet (SSGRR 2002w), 2002.

7. Fallside, D. C.: XML Schema Part 0: Primer. `http://www.w3.org/TR/xmlschema-0/`, W3C Recommendation, May 2001.

8. Halevy, A. Y., Ives, Z. G., Mork, P., Tatarinov, I.: Piazza: Data Management Infrastructure for Semantic Web Applications. In Proceedings of the 12th International World Wide Web Conference (WWW2003), pages 556-C567, 2003.

9. Klein, M. C. A.: Interpreting XML Documents via an RDF Schema Ontology. In Proceedings of the 13th International Workshop on Database and Expert Systems Applications (DEXA 2002), pages 889-C894, 2002.

10. Alashqur, A. M., Su, S. W., and Lain., H.: OQL: A Query Language for Manipulating Object-Oriented Databases. In Proc. Int. Conf. on Very Large Data Bases, pages 433–442, 1989.

66

11. Seaborne, A.: RDQL—A Query Language for RDF. `http://www.w3.org/Submission/RDQL/`, W3C Member Submission, January, 2004.

12. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., and Scholl, M.: RQL: A Declarative Query Language for RDF. In The 11th International World Wide Web Conference, 2002.

13. The SeRQL Query Language. `http://www.openrdf.org/doc/users/ch05.html`.

14. Lakshmanan, L. V. and Sadri, F.: Interoperability on XML Data. In Proceedings of the 2nd International Semantic Web Conference (ICSW03), 2003.

15. Patel-Schneider, P. F. and Siméon, J.: The Yin/Yang web: XML syntax and RDF Semantics. In Proceedings of the 11th International World Wide Web Conference (WWW2002), pages 443C-453, 2002.

16. Connolly, D., van Harmelen, F., Horrocks, I., McGuiness, D. L., Patel-Schneider, P. F., and Stein., L. A.: DAML+OIL Reference Description. `http://www.w3.org/TR/daml+oil-reference`, W3C Note, December 18, 2001.

17. Smith, M. K., Welty, C., and McGuinness, D. L.: OWL Web Ontology Language Guide. `http://www.w3.org/TR/owl-guide`, W3C Recommendation, February 10, 2004.

18. Heflin, J. and Hendler, J.: Semantic Interoperability on the Web. In Extreme Markup Languages 2000, pages 111–120, 2000.

19. Heflin, J. and Hendler, J.: SHOE: A Knowledge Representation Language for Internet Applications. Technical Report CS-TR-4078, Dept. of Computer Science, University of Maryland at College Park, 1999.

20. Vdovjak, R. and Houben, G.: RDF Based Architecture for Semantic Integration of Heterogeneous Information Sources. In International Workshop on Information Integration on the Web, pages 51–57, April 2001.

21. Suciu, D.: Distributed Query Evaluation on Semistructured Data. ACM Transactions on Database Systems, 2002.

22. Clark, J. and Makoto, M.: Relax NG Specification. `http://www.relaxng.org/spec-20011203.html`, December 3, 2001.

23. Berners-Lee, T.: Notation 3—Ideas about Web Architecture. `http://www.w3.org/DesignIssues/Notation3.html`, November 2001.

24. Bray, T., Paoli, J., Sperberg-McQueen, C. M., and Maler, E.: Extensible Markup Language. `http://www.w3.org/TR/REC-xml`, W3C Recommendation, February 2004.

25. Document Object Model (DOM) Level 1 Specification. Apparao, V., Byrne, S., Champion, M., Isaacs, S., Le Hors, A., Nicol, G., Robie, J., Sharpe, P., Smith, B., Sorensen, J., Sutor, R., Whitmer, R., and Wilson, C.: `http://www.w3.org/TR/REC-DOM-Level-1/`, W3C Recommendation, October, 1998.

VITA

NAME: Feihong Hsu

EDUCATION: B.S., Mathematics and Computer Science,

University of Illinois at Urbana-Champaign,

Urbana, Illinois, 2000

M.S., Computer Science,

University of Illinois at Chicago,

Chicago, Illinois, 2004

HONORS: Outstanding TA of the Year Award, 2003

PUBLICATIONS: Reed, D., John, S., Aviles, R., and Hsu, F.: CFX:

Finding Just the Right Examples for CS1. In

ACM Special Interest Group on Computer Science

Education (SIGCSE), Norfolk, Virginia, Mar. 2004.