The State of the Art on Representation and Query Languages for Semistructured Data

Jeen Broekstra
Christiaan Fluit
Frank van Harmelen
{jeen.broekstra, christiaan.fluit, frank.van.harmelen}@aidministrator.nl

Summary. During the past few years, research into representation of semistructured (meta)data has taken an enormous flight. This document presents an overview of a couple of the most promising approaches in this field, such as XML and RDF. We also investigate the current research into query languages tailored to semistructured data , by first examining what properties such a language should have and then reviewing the most prominent existing approaches, such as XSLT, XQL, XML-QL, Lorel and the proprietary rule format used by aidministrator to describe constraints on the structure and contents of tree-structured documents, and we take a look at the efforts undertaken to specify a query language specifically for RDF.

Document Id.	deliverable 8
Project	EU-IST On-To-Knowledge IST-1999-10132
Issuer	aidministrator nederland b.v.
Date	10th August 2000
Status	Final
Distribution	Public

Copyright © 2000 aidministrator nederland b.v.

On-To-Knowledge Consortium

This document is part of a research project partially funded by the IST Programme of the Commission of the European Communities as project number IST-1999-10132. The partners in this project are: Vrije Universiteit Amsterdam VUA (coordinator, NL), University of Karlsruhe (Germany), Schweizerische Lebensversicherungs- und Rentenanstalt/Swiss Life (Switzerland), British Telecommunications plc (UK), CognIT a.s. (Norway), EnerSearch AB (Sweden), AIdministrator Nederland BV (NL).

Vrije Universiteit Amsterdam (VUA)
Division of Mathematics and Informatics W&I
De Boelelaan 1081a
1081 HV Amsterdam
The Netherlands
Tel: +31 20 4447718, Fax: +31 20 872 27 22
Contactperson: Dieter Fensel
E-mail: dieter@cs.vu.nl

University of Karlsruhe
Institute AIFB
Kaiserstr. 12
D-76128 Karlsruhe
Germany
Tel: +49 721 608392, Fax: +49 721 693717
Contactperson: R. Studer
E-mail: studer@aifb.uni-karlsruhe.de

Schweizerische Lebensversicherungs- und Rentenanstalt / Swiss Life
Swiss Life Information Systems Research Group
General Guisan-Quai 40
8022 Zürich
Switzerland
Tel: +41 1 284 4061, Fax: +41 1 284 6913
Contactperson: Ulrich Reimer
E-mail: Ulrich.Reimer@swisslife.ch

British Telecommunications plc
BT Adastral Park
Martlesham Heath
IP5 3RE Ipswich
United Kingdom
Tel: +44 1473 605536
Fax: +44 1473 642459
Contactperson: John Davies
E-mail: John.nj.Davies@bt.com

CognIT a.s
Busterudgt 1.
N-1754 Halden
Norway
Tel: +47 69 1770 44, Fax: +47 669 006 12
Contactperson: Bernt. A.Bremdal
E-mail: bernt@cognit.no

EnerSearch AB
Carl Gustafsväg 1
SE 205 09 Malmö
Sweden
Tel: +46 40 25 58 25; Fax: +46 40 611 51 84
Contactperson: Hans Ottosson, Fredrik Ygge
E-mail: hans.ottosson,fredrik.ygge@enersearch.se

AIdministrator Nederland BV
Julianaplein 14B
3817 CS Amersfoort
The Netherlands
Tel: +31 33 4659987, Fax: +31 33 4659987
Contactperson: Jos van der Meer
E-mail: Jos.van.der.Meer@aidministrator.nl

aidministrator nederland b.v.
julianaplein 14b
3817 cs amersfoort
the netherlands
tel. (+31)(0)33 4659987
fax. (+31)(0)33 4659987
http://www.aidministrator.nl

Acknowledgements

The research reported here was carried out in the course of the On-To-Knowledge project. This project is partially funded by the IST Programme of the Commission of the European Communities as project number IST-1999-10132. The partners in this project are: Vrije Universiteit Amsterdam VUA (coordinator, NL), University of Karlsruhe (Germany), Swiss Life (Switzerland), BT plc (UK), CognIT a.s. (Norway), EnerSearch AB (Sweden), AIdministrator Nederland BV (NL).

The authors of this report wish to thank Arjohn Kampman, Herko ter Horst and Stefan Decker for their most helpful feedback and comments on earlier versions of this report.

Contents

1 Representation languages **3**
1.1 Introduction . . . 3
1.2 HTML . . . 4
1.3 Ontobroker . . . 4
1.3.1 Remarks and observations . . . 6
1.4 SHOE . . . 8
1.4.1 Differences with Ontobroker . . . 10
1.4.2 Remarks and observations . . . 10
1.5 XML . . . 11
1.5.1 XML Schema . . . 12
1.6 RDF . . . 13
1.6.1 RDF Schema . . . 14
1.6.2 Remarks and observations . . . 16
1.7 A comparison of approaches . . . 16
1.7.1 Supported by Web technology . . . 16
1.7.2 Avoiding duplication of data . . . 16
1.7.3 Separation of data and meta data . . . 17
1.7.4 Data models . . . 18
1.7.5 Ontologies . . . 18
1.7.6 Conclusions . . . 18

2 Query languages **21**
2.1 Introduction . . . 21
2.2 General properties of QLs for semistructured data . . . 21
2.2.1 Path expressions . . . 22
2.3 Why not just SQL? . . . 22
2.4 Querying XML . . . 23
2.4.1 XSL . . . 23
2.4.2 XQL . . . 25
2.4.3 XML-QL . . . 26
2.4.4 Conclusion . . . 27
2.5 Lore . . . 27
2.5.1 OEM . . . 27
2.5.2 Lorel . . . 28
2.5.3 Architecture . . . 30
2.5.4 Remarks and observations . . . 30
2.6 aidministrator's visual rule format . . . 31
2.6.1 The logical structure . . . 31
2.6.2 The visual structure . . . 32
2.6.3 Comparison with requirements . . . 33
2.7 Querying RDF . . . 34
2.7.1 The SQL/XQL style approach . . . 35

2.7.2 The declarative approach . . . 36
2.7.3 RQL in depth . . . 39
2.8 A comparison of approaches . . . 41

3 Conclusions **43**

Chapter 1

Representation languages

1.1 Introduction

The lack of machine-accessible semantics is a major barrier to the development of more intelligent document processing on the Web. Current HTML markup is used only to indicate the structure and layout of documents, but not the semantics of the information presented.

The On-To-Knowledge project aims to use the power of ontologies to facilitate navigation and querying of semistructured data . However, while ontologies provide essential background information on a domain, factual instances of concepts still need to be recognized in the available data. In other words: a representation language for semistructured data is required.

What is semistructured data ?

Semistructured data can be explained as ”schemaless” or ”self-describing”, indicating that there is no seperate description of the type or strucutre of data. This is in contrast with structured approaches (such as, e.g. databases), where usually the structure of the data is described first in a seperate schema, and then instances of this schema are created. In semistructured data we directly describe the data using a simple syntax. Usually, this means semistructured data consists of simple label-value pairs (cf. [Abiteboul et al., 1999]), for example:

```
{name: "Frank", tel: 47731, email: "frankh@cs.vu.nl"}
```

When we allow the values themselves to be structures, we get a graph data model with a tree-like structure:

```
{name: {first: "Frank", last: "van Harmelen"},
 tel: 47731,
 email: "frankh@cs.vu.nl"
}
```

No uniqueness constraint is placed on the labels (as one might expect in more conventional approaches), allowing multiple label-value pairs with the same label.

One of the main strenghts of semistructured data is the ability to accomodate variations in structure. These variations typically consist of missing data, duplicated fields, or minor changes in representation.

Structure of this chapter

Several methods have been developed (or are currently under development) to enable semantic annotation of Web documents. Of these, we will discuss two approaches from the AI community (Ontobroker, section 1.3, and SHOE, section 1.4), and the suggested approaches by the W3C,

namely HTML (section 1.2), XML (section 1.5) and RDF (section 1.6). The approaches by the W3C combine Web-based technology with suggested methodology from the Databases community (such as, for example, schema definitions).

1.2 HTML

This section is an excerpt from [van Harmelen and Fensel, 1999].

META tags

Historically, the first attempt at representing semantic aspects inside Web documents are the HTML META-tags. Their intended use is limited to stating global properties that apply to the entire document, for example:

```
<HEAD>
    <META NAME="author" CONTENT="Frank">
</HEAD>
```

This expresses that the author of the entire document is Frank.

In its original form, this mechanism is rather inflexible and only allows assertion of global properties that hold for the entire document.

`SPAN` and `DIV` elements

According to the HTML 4.0 specification [Raggett et al., 1999] the `SPAN` element "is a generic container of any text element offering a generic mechanism for adding structure to documents". Using the standard `CLASS` attribute one can write semantic annotations such as:

```
<BODY>
This page is written by
  <SPAN CLASS="author">Frank van Harmelen</SPAN>.
  <SPAN CLASS="location">
    His tel.nr. is <SPAN CLASS="tel">47731</SPAN>,
    room nr. <SPAN CLASS="room">T3.57</SPAN>
  </SPAN>
</BODY>
```

Although intended for specifying layout (enabling simple generic stylesheet mechanisms), the HTML 4.0 specification already suggests the use of the `SPAN` element to express semantic structure of a document, so this use of the `SPAN` tag should not be considered as inappropriate.

The `DIV` element works practically identically to the `SPAN` element. In fact, the only difference between the two is that `SPAN` is an inline element whereas `DIV` is a block element. This is a parser-level distinction that is not really important for our discussion. A detailed explanation of the issue can be found in [Raggett et al., 1999].

1.3 Ontobroker

Ontobroker [Fensel et al., 1998, Fensel et al., 1999] is a project from the University of Karlsruhe dedicated to building a set of languages and tools for enabling and enhancing query and inference services on the web. One of these languages provides the ability to semantically annotate ontological information present in existing webpages. This language will be the main topic of this section. At `http://ontobroker.aifb.uni-karlsruhe.de/demos.html` some example Ontobroker query services can be found.

Ontologies play a major role in the Ontobroker project. Within a certain community, referred to as an *ontogroup*, an ontology is used to model a shared view on the relevant entities in that domain. Such an ontology consists of a hierarchy of classes, attributes, relations and logical axioms describing the domain, and is described in Frame Logic [Kifer et al., 1995]. This ontology is essentially defined centrally, i.e. all people within the community need to use this ontology in order to participate; there is no mechanism for making personal refinements or views.

This is a straightforward example of an Ontobroker ontology, describing two classes and their typed attributes:

```
Object[].
Person::Object.
Researcher::Person.

Person[
  name =>> STRING].

Researcher[
  affiliation =>> Organization;
  cooperatesWith =>> Researcher].

Organization[
  ...
].

...
```

Such an ontology would probably be stored at the server providing the query and inference services.

In order to provide instances for the ontology classes, attributes and relations, an annotation language is offered. The two design criteria for the design of this language were:

1. It should integrate smoothly in HTML, the current de facto standard for web pages.
2. It should reuse existing information in these webpages as much as possible, thus preventing duplication of information (as needed in some other approaches like SHOE and RDF).

The annotation language is called HTML^{A} and extends the HTML language with an extra `onto` attribute for the HTML anchors (`A` elements). For declaring the Researcher instance "Frank van Harmelen", all one has to do is putting a statement like

```
<a onto="http://www.cs.vu.nl/~frankh/:Researcher">
```

somewhere in *any* document. This asserts the existance of an instance of class Researcher with object identifier `http://www.cs.vu.nl/~frankh/`. The name attribute of this Researcher instance can then be declared as follows:

```
<a onto="http://www.cs.vu.nl/~frankh/[name='Frank van Harmelen']">
```

The value of an attribute can be a literal such as a `STRING` or another object identifier. We provide no examples of defining and instantiating relations, since we dit not encounter any specifications nor examples of how to do that.

This way of annotating documents satisfies the first design criterion (smooth integration in HTML), since all `onto` attributes are safely ignored by the browser. In order to fulfil the second criterion (prevention of duplication of information), macros are available that allow for shorter and better maintainable annotations. For example, it is often the case that an object identifier in a statement is equal to the URL of the page containing the statement, e.g. a statement in Frank's homepage mentioning that "Frank van Harmelen" is a Researcher with the URL of that homepage as an object identifier. Using the `page` macro, this can be abbreviated to:

```
<a onto="page:Researcher">
```

When the annotations are extracted, all `page` macros will be replaced by the URL of the page. Other available macros are:

- `tag` and `name`, which are both replaced by the URL of the page appended with a hash and the value of the `name` attribute of the same anchor.
- `href`, which is replaced by the value of the `href` attribute of the same anchor.
- `body`, which is replaced by the text content of the anchor, i.e. the text inbetween `<a ...>` and `</a>`.

These macros can be used when declaring instances as well as attribute values, e.g. as in

```
Hi, my name is <a onto="page[name=body]">Frank van Harmelen</a>
```

Although the ontology might specify that an attribute has either a literal or an object identifier as its value, both will be accepted by annotation processors, since in reality it may happen quite often that the object identifier of a real world object is not yet established or known. Consider for example a Publication class with an author attribute of type Person: it may be that there is no object representing the author, so that a string value is the best you can do.

Another problem with object identification is the fact that URLs are typically used as identifiers. Although it makes sense in a web context to use them for this purpose, the problem is that URLs are often not unique: the same webpage may be reachable through several different URLs, e.g. `http://www.somehost.com/` and `http://www.somehost.com/index.html`. This is a major issue when using macros instead of explicit object identifiers. There is no obvious solution for this problem, since the rules for equality of URLs may differ from host to host. This problem is acknowledged but ignored by the Ontobroker team.

The Ontobroker tool set consists of a number of various tools. A crawler has been made that gathers annotated pages from the Internet, extracts the annotations and stores them in a database. A search and inference engine is able to execute F-logic queries on this database, in terms of ontological entities. This engine uses knowledge from the ontology to enhance the query and produce more results, e.g. by using subclass relationships, transitivity of relations, etc. The query language also allows meta-level queries such as “give me all available attributes of the Researcher class”. Two query front-ends are available, one allowing expert users to directly input this F-logic query, the other one assisting naive users with formulating such a query using forms and a graphical representation of the class hierarchy. The latter interface makes the query engine accessible for people who are not familiar with F-logic as well as for people who have no knowledge about the ontology in advance.

A simple example F-logic query on the ontology discussed earlier is:

```
FORALL FH, R, NAME <-
    FH:Researcher[name->>''Frank van Harmelen''] and
    R:Researcher[name->>NAME;cooperatesWith->>FH].
```

which retrieves the object identifiers and names of all Researchers cooperating with “Frank van Harmelen”.

1.3.1 Remarks and observations

An advantage of HTML^A is clearly the fact that it integrates into existing webpages, preventing the need for a seperate metadata storage with its associated problems like information duplication and – consequently – maintenance.

However, it remains to be seen whether a separate storage really is a problem within the Ontoknowledge project, especially when the metadata is generated automatically. Additionally, it might not always be possible to adapt the original source, e.g. when the owners do not give

permission for it or when another document format is used such as PDF or Word's proprietary format, ruling out the approach offered by HTML^A.

Furthermore, the mechanisms offered by HTML^A for reusing existing information are rather limited. Two kinds of problems may occur. First, a fragment of a webpage might mention a lot of information about a single object at once, e.g.:

> D. Fensel, S. Decker, M. Erdmann, R. Studer.
> *Ontobroker: The Very High Idea.*
> In: *Proceedings of the 11th International Flairs Conference (FLAIRS-98)*, Sanibal Island, Florida, May 1998.

This information could be used for introducing a Publication instance and immediately filling in its attributes. In approaches like XML this information would be very straightforward to model, e.g. by a `PUBLICATION` element with `AUTHOR` child elements, etc. However, due to the lacking capability of nesting ontological statements (anchors are not allowed to nest) the object identifiers have to be repeated over and over again. Using lots of macros, the best one can do would probably be (omitting layout markup):

```
<a name="FLAIRS-98" onto="tag[author=body]">D. Fensel</a>,
<a name="FLAIRS-98" onto="tag[author=body]">S. Decker</a>,
<a name="FLAIRS-98" onto="tag[author=body]">M. Erdmann</a>,
<a name="FLAIRS-98" onto="tag[author=body]">R. Studer</a>.
<a name="FLAIRS-98" onto="tag:ConferencePaper"
onto="tag[title=body]>Ontobroker: The Very High Idea.</a>
In: <a name="FLAIRS-98" onto="tag[proceedingsTitle=body]">
Proceedings of the 11th International Flairs Conference
(FLAIRS-98)</a>,
Sanibal Island, Florida,
May <a name="FLAIRS-98" onto="tag[publicationYear=body]">1998</a>.
```

This problem may be solved by using tags that do allow for nesting, such as the `SPAN` and `DIV` tags, and defining a proper semantics for the nesting of ontological statements.

However, even offering nesting of annotation will not solve the second problem: information may as well be spread among various parts of a document or even several documents, e.g. a document containing information about employees working on a project might present their names and contact information in one table, their role in the project in another table, etc. When asserting this information as object instances and attributes, every annotation again needs to refer to the original object by its object identifier. This problem is caused by the fact that semantic annotations are inserted in documents originally formatted for layout purposes. In approaches like SHOE or XML, where presentational aspects are not present or are taken care of separately, this data would be modeled differently and the problem would never occur, at least not for such simple cases.

Finally three remarks about the framework as a whole. First, the whole framework, including the annotation language, lacks the notion of the origin of the information: once entered in the database, it cannot be retrieved where the original information came from and, equally important, who made the claims. Especially in a broad Internet context but also in an intranet context this information may be highly relevant under certain circumstances.

Second, the choice to let a group of people decide on a shared ontology does not allow for the combination of knowledge that is on web pages within different ontological domains. Yet this kind of combination may in practice be desirable. After all, there are many pages out there that do not fit neatly into one single ontology.

Third, there is no considerable industry support for this framework and its annotation language, nor any support from standardization organizations. This will make it difficult to persuade people to apply it.

1.4 SHOE

SHOE [Heflin et al., 1998, Heflin et al., 1999], which stands for *Simple HTML Ontology Extensions*, is an extension to HTML that offers a representation language for describing ontologies and their populations. Ontologies consist of a hierarchy of classes with attributes and relations, as well as simple rules for doing inferences on the ontological information. These ontologies offer the basis for ontological markup that can be embedded in HTML pages.

This may sound very similar to Ontobroker. There are however a few very fundamental differences. We will first give some examples of how SHOE is used, followed by an overview of these differences between the two frameworks.

The following is an example of a simple SHOE ontology specification, embedded inside an HTML document:

```
<HTML>
<HEAD>
    <META HTTP-EQUIV="SHOE" CONTENT="VERSION=1.0">
    <TITLE>Example Ontology</TITLE>
</HEAD>
<BODY>
    <!-- create a new ontology -->
    <ONTOLOGY ID="person-ontology" VERSION="1.0"
     DESCRIPTION="Simple example ontology">
    <USE-ONTOLOGY ID="base-ontology" VERSION="1.0" PREFIX="base"
     URL="http://www.cs.umd.edu/projects/plus/SHOE/base.html">

        <!-- definition of a few classes -->
        <DEF-CATEGORY NAME="Person" ISA="base.SHOEEntity">
        <DEF-CATEGORY NAME="Researcher" ISA="Person">

        <!-- definition of a few relations -->
        <DEF-RELATION NAME="name">
            <DEF-ARG POS="1" TYPE="Person">
            <DEF-ARG POS="2" TYPE="STRING">
        </DEF-RELATION>
        <DEF-RELATION NAME="cooperatesWith">
            <DEF-ARG POS="1" TYPE="Researcher">
            <DEF-ARG POS="2" TYPE="Researcher">
        </DEF-RELATION>
    </ONTOLOGY>
    ...
    <!-- regular content comes here -->
    ...
</BODY>
</HTML>
```

The `META` element indicates the SHOE version that will be used for specifying this ontology. The ontology will be known as `person-ontology`, and is an extension of `base-ontology`, the universal root ontology in the hierarchy of ontologies. The `PREFIX` is a string that will be used to prefix all constructs from `base-ontology` with, similar to the namespace concept in XML. SHOE allows inclusion of multiple ontologies, thus providing the possibility of multiple inheritance in the ontology hierarchy.

The ontology defines a class `Person` with subclass `Researcher`. `Person` is a subclass of class `SHOEEntity`, the universally heighest class defined in `base-ontology`.

Furthermore two typed relations are defined in this ontology. Relations can be n-ary and are used to state *claims*: facts proposed by a certain instance. SHOE emphasizes that last aspect since

facts from different instances may contradict. This enables an agent who is using the claims to decide what to do with it. One of these relations makes use of the built-in STRING datatype. Other built-in datatypes are NUMBER, TRUTH and DATE.

In addition, SHOE ontologies may also define production rules that derive information from the existing claims, in the form of Horn clauses without negation. We will not cover that aspect however because it falls outside the scope of the search for an appropriate metadata representation format.

Once we have defined this ontology, we can use it to add extra annotations to existing webpages. This is an example of an HTML page containing a SHOE instance:

```
<HTML>
<HEAD>
    <META HTTP-EQUIV="SHOE" CONTENT="VERSION=1.0">
    <TITLE>Frank's Page</TITLE>
</HEAD>
<BODY>
    <INSTANCE KEY="http://www.cs.vu.nl/~frankh/">
        <USE-ONTOLOGY ID="person-ontology"
         URL="http://www.cs.vu.nl/~frankh/person.html"
         VERSION="1.0" PREFIX="person">

        <CATEGORY NAME="person.Researcher">

        <RELATION NAME="person.name">
            <ARG POS="1" VALUE="me">
            <ARG POS="2" VALUE="Frank van Harmelen">
        </RELATION>
        <RELATION NAME="person.cooperatesWith">
            <ARG POS="TO" VALUE="http://www.cs.vu.nl/~dieter/>
        </RELATION>
    </INSTANCE>
...
<!-- regular content comes here -->
...
</BODY>
</HTML>
```

The META element again indicates which SHOE version will be used. The INSTANCE element declares a new object which has a URL as its object identifier (KEY). SHOE proposes but does not require that each KEY begins with the URL of the page that contains the corresponding INSTANCE. A KEY may be extended with a hash and an additional string in case the page contains multiple instances. The instance will use the person-ontology for its assertions. All constructs from that ontology will have the prefix person.

The instance first declares itself to be of the Person category. Multiple inheritance is allowed here, including inheriting from categories defined in other ontologies, since all names used are uniquely defined through their prefixes.

Furthermore, two relations are defined. The first relation claims that the name of the instance is equal to the string "Frank van Harmelen". Several shortcuts can be used when defining a relation. One is used here: the value me expands in the value of the KEY of the INSTANCE in which it is used.

The second relation states that this instance "cooperatesWith" the instance with KEY value http://www.cs.vu.nl/~dieter/. It makes use of another shortcut construction: in case the relation is binary, the attributes POS="1" and POS="2" can be changed to POS="FROM" and POS="TO". Additionally, if and only if this alternative notation is used and one of the POS attributes has the value "me", it may be omitted.

Finally, the instance declaration is closed with `</INSTANCE>`, after which the regular content of the document follows. The fact that the SHOE annotations are in the same file as the regular content does not matter since a HTML parser ignores all elements that it does not recognize. Note that all SHOE declarations are placed inside attributes of the SHOE elements, not in regular element text, so that they really are completely unvisible when this page is viewed in a browser.

The SHOE project also has a number of tools that deal with SHOE-enabled pages. A crawler has been made that gathers webpages with SHOE instances, extracts them and stores them into a database. When it encounters ontologies that it has not encountered before, it loads and stores them as well. Additionally, a tool called the Knowledge Annotator has been developed which assists its users with creating SHOE instances without requiring knowledge about its syntax or the ontology in advance. However, there seems to be no support for editing and maintaining the ontologies themselves. Various other tools have been developed that query the database and present the results.

1.4.1 Differences with Ontobroker

Although the kind of ontological entities in Ontobroker and SHOE are much alike, there are some key differences in the way they are used:

- Ontobroker requires a centralized ontology, whereas SHOE allows everyone to extend existing ontologies or define new ontologies from scratch.
- SHOE ontologies are embedded in webpages, whereas Ontobroker stores them separately, centrally and using a completely different representation language.
- HTML^A, Ontobroker's annotation language for populating an ontology, tries to reuse as much data as possible from the documents containing the annotations, whereas SHOE's ontological annotations need to duplicate this information.
- SHOE does not have an explicit *attribute* representation, they need to be simulated by relations. This has two consequences:
    1. These "attributes" are always optional, since cardinality constraints on relations cannot be specified.
    2. Any instance can *claim* the attribute value of any other instance.

1.4.2 Remarks and observations

One usability problem of SHOE seems to be that it is a rather verbose annotation language. However, this may be overcome in two different ways:

1. Annotations may be generated by special purpose editing tools, hiding the syntactic details of the annotation language. In fact, one such tool has already been made: the Knowledge Annotator.
2. The problem completely vanishes when the data is generated automatically instead of edited manually, as may be the case within the Ontoknowledge project.

Another problem of SHOE is that it, in contrast to Ontobroker, requires you to duplicate the information that is already present in webpages. This problem remains even when using annotation tools, but, on the other hand, is less relevant when the information is (semi)automatically generated.

Furthermore, allowing for decentralized ontology definitions raises the question how these ontologies are to be used effectively, since the implicit semantics of the entities in the ontology still must be known in order to query them reasonably, offer data exchange facilities between domains, etc.

Finally, SHOE also has no considerable standardization and industry support, making it unattractive for people to rely on it.

1.5 XML

The **Extensible Markup Language (XML)** [Bray et al., 1998] describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them. XML is an application profile or restricted form of SGML, the Standard Generalized Markup Language [ISO8879, 1986]. By construction, XML documents are conforming SGML documents.

XML, while named a markup language, is actually a markup meta language. It allows one to define a set of markup tags, which can be chosen to reflect the domain specific semantics of the information, rather than merely its layout and structure (as is the case in, for example, HTML).

An XML document consists of a properly nested set of open and close tags, where each tag can have a number of attribute-value pairs. The vocabulary of the tags and their allowed combination is not fixed, but can be defined per application of XML. An example XML document is:

```
<?xml version="1.0"?>
<body>
    This page is written by
    <author>Frank van Harmelen</author>.
    <location>
    His tel.nr. is
        <tel>47731</tel>,
            and his room number is
        <room>T3.57</room>.
    </location>
</body>
```

From the indentation of the above example, it is easy to see that the basic data model of XML is a labeled tree, where each tag corrsponds to a labeled node in the data model, and each nested subtag is child in the tree.

The intended structure of XML documents within a particular domain, i.e. the allowed labels and the way in which they can be nested, is described in a Document Type Declaration (DTD), which expresses in a grammar-like formalism which allowed sequences and nestings of tag are allowed. For example, a DTD for the above XML file would possibly look like this:

```
<?xml version="1.0"?>
<!ELEMENT body (author, location, #PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT location (tel*, room?, institution?, city?, #PCDATA)>
<!ELEMENT tel (#PCDATA)>
<!ELEMENT room (#PCDATA)>
<!ELEMENT institution (#PCDATA)>
<!ELEMENT city (#PCDATA)>
```

DTDs offer very limited expressivity. For example, it is only possible to describe the legal lexical nesting of elements, no other relationships between elements can be expressed. Also, virtually no typing of values is possible. These shortcomings make DTDs unsuited for real schema modeling (in fact, it was always intended as a ”stop-gap” method until a more suitable schema language was developed). Developments are well underway at the W3C to replace DTDs with XML Schema definitions [Thompson et al., 2000, Byron and Malhotra, 2000].

In practice, XML is being used for a number of rather different purposes:

- as a serialization syntax for other markup languages. For example, the SMIL multi-media markup language [Hoschka, 1998] uses the XML format for its markup syntax. Syntactically, SMIL is simply a particular XML DTD. The real value of SMIL lies in the fact that

there exists a common understanding of the intended meaning of the elements of that particular DTD. However, it is important to realize that the DTD only specifies the syntactic conventions, and that any such intended semantics remain outside the realm of hte XML specification.

- as semantic markup of Web-pages (as seen in the sample XML document above).
- as a uniform data-exchange format. The above example can also be a data object transferred between two applications. Again, only the syntactic structure of XML is enforced; the intended meaning of the various elements is entirely implicit in the XML document.

Because the intended semantics of XML tagging is always implicit (since neither the XML definition itself nor the DTD specify anything but syntax), only the first usage of XML is in real accordance with the original goal of the language.

1.5.1 XML Schema

XML Schemas are a means for defining constraints on valid XML documents. The **XML Schema Specification** is divided into two parts: in [Thompson et al., 2000] Structures are described, and in [Byron and Malhotra, 2000] Datatypes are described. A human readable explanation on XML Schemas can be found in [Fallside, 2000].

XML Schemas have the same purpose as DTDs, but they provide several significant improvements:

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

<schema xmlns="http://www.w3.org/1999/XMLSchema">
    <complexType name="address">
        <element name ="name" minOccur="1" maxOccur="1" type="string"/>
        <element name="street" minOccur="1" maxOccur="2" type="string"/>
        <element name="city" minOccur="1" maxOccur="1" type="string"/>
        <element ref="zip" minOccur="1" maxOccur="1"/>
    </complexType>

    <element name="zip" type="zipCode"/>

    <simpleType name="zipCode" base="string">
        <pattern value="[0-9]{5}(-[0-9]{4})?"/>
    </simpleType>
</schema>
```

Figure 1.1: an example XML schema

- XML Schema definitions are XML documents themselves. For example in figure 1.1 the schema definition for an 'address' tag is given. The schema itself is in XML, whereas a traditional DTD would provide such a definition in an external other language. The advantage is that all tools developed for XML (e.g. validation and rendering tools) can immediately be applied to Schemas as well.
- XML Schema provides a rich set of datatypes that can be used to define the values of elementary tags.
- XML Schema provides much richer means for defining nested tags (i.e., tags with subtags).

- XML Schema provides the namespace mechanism to combine XML documents with heterogeneous vocabulary.
- XML Schema provides a mechanism to define new datatypes, by extending or constraining existing datatypes to form new subtypes. For example in figure 1.1 we see a simple datatype `zipCode` which is defined by constraining the basic datatype `string` to allow only certain patterns as values.

A more detailed discussion of XML Schema can be found in [Klein et al., 2000].

At first sight XML Schema may seem an attractive language for ontology definitions with datatyping and type extension mechanisms. However, the conclusion of [Klein et al., 2000] is that XML Schema is unsuited for this purpose and should only be used for describing the syntactic structure of documents (a conclusion supported by the developers of XML Schema themselves [H. Thompson, personal communication]).

1.6 RDF

The **Resource Description Framework (RDF) Model and Syntax specification** [Lasilla and Swick, 1999] describes a foundation for processing meta data. RDF provides interoperability between applications that exchange machine-understandable information on the Web. Basically, RDF defines a data model for describing semantics of data in a syntax-independent way. The data model consists of three basic object types:

- **Resources**: a resource is anything that is addressable by a URI. For example, any web page or specific part of a web page is a resource.
- **Properties**: a property is a specific attribute or relation used to describe a resource.
- **Statements**: a statement specifies for a particular resource the value of a property.

So, RDF statements consist of ordered triples of the form $\langle resource, property, value \rangle$.

A simple example of an RDF triple is:

```
Author(http://www.cs.vu.nl/~frankh) = Frank
```

where the URL is a pointer to a resource, the Author predicate a property, and "Frank" the value of that property with respect to this particular resource.

RDF allows second order statements by means of reification over the value of a triple. For example:

```
Claim(Dieter) = (Author(http://www.cs.vu.nl/~frankh) = Frank)
```

states that Dieter claims that Frank is the author of the document at the given URL.

RDF uses a directed, asymmetrical, labeled graph as its data model, where properties are edges and resources and values are nodes.

[Lasilla and Swick, 1999] also specifies an XML syntax for RDF descriptions. The XML serialization of the statement about Frank's homepage would look like this:

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:d="http://www.description.org/schema/">
    <rdf:Description rdf:about="http://www.cs.vu.nl/~frankh">
        <d:Author>Frank</d:Author>
    </rdf:Description>
</rdf:RDF>
```

The namespace prefix 'd' refers to a specific namespace chosen by the author of this RDF expression. It is defined in the XML namespace declaration given in the rdf:RDF element. Namespace declarations can also be given on particular Description elements.

In practice, RDF is already being used in several areas. For example, the Mozilla Open Directory Project[1], an open source initiative to create a hierarchical, browsable directory of Web pges, dumps both its structure and content in RDF. Another example is PICS[2]. The PICS specification enables labels (metadata) to be associated with Internet content. It was originally designed to help parents and teachers control what children access on the Internet, but it also facilitates other uses for labels, including code signing and privacy.

1.6.1 RDF Schema

RDF Schema (RDFS) is used to define the structure of the meta data that are used to describe resources. The modeling primitives provided by RDF are very basic — they correspond to binary predicates of ground terms, where, however, the predicates may be used as terms as well. The **RDF Schema specification** [Brickley and Guha, 2000] defines further modeling primitives in RDF. Examples are class and subclass relationships, and domain and range restrictions for properties. With these extensions RDF Schema enables the definition of a typing schema in an object-oriented manner. In figure 1.2 we see a simple example of (a part of) an RDF Schema, which describes a class `Researcher` and a property `author`. Note how primitives from both the rdf and the rdfs namespace are used to define a schema.

```
<rdfs:Class rdf:id="Researcher">
    <rdfs:subClassOf="#Person"/>
</rdfs:Class>

<rdf:Property rdf:id="author">
    <rdfs:domain rdf:Resource="#Publication"/>
    <rdfs:range rdf:Resource="#Researcher"/>
</rdf:Property>
```

Figure 1.2: An example RDF Schema

In this section, we will briefly discuss the overall structure of RDFS and its main modeling primitives (see also [Horrocks et al., 2000]).

The data model of RDF Schema

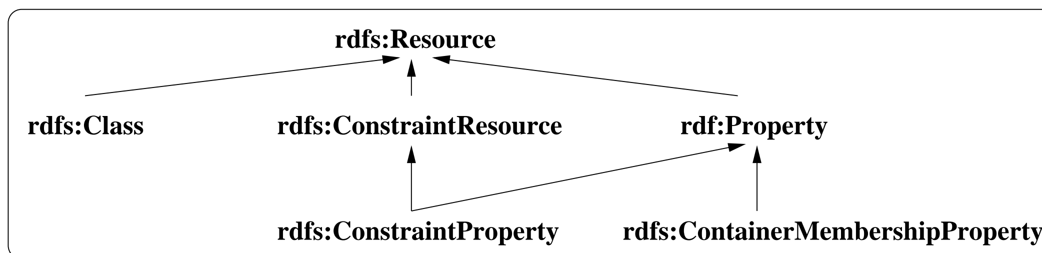

Figure 1.3: The subclass-of hierarchy of modeling primitives in RDFS

[1] http://www.dmoz.org
[2] http://www.w3.org/PICS

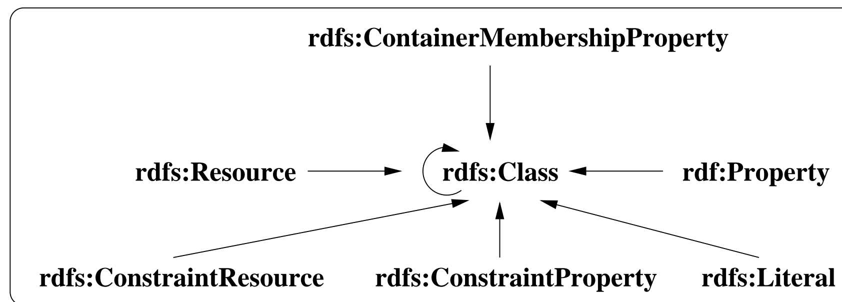

Figure 1.4: The instance hierarchy of modeling primitives in RDFS

Figure 1.3 pictures the *subclass-of* hierarchy of RDFS. The *instance-of* hierarchy is depicted in figure 1.4 (cf. [Brickley and Guha, 2000]). The "rdf" prefix refers to the RDF name space (i.e. primitives with this prefix are already defined in RDF) and "rdfs" refers to new primitives defined by RDFS. If we take a closer look at these figures a number of peculiarities can be noticed, for example:

- rdfs:Class is an instance of rdfs:Class.
- rdfs:Class is a subclass of rdfs:Resource and rdfs:Resource is an instance of rdfs:Class

It is noted by several authors ([Nejdl et al., 2000, Broekstra et al., 2000, Horrocks et al., 2000]) that the object-meta model used for RDFS is non-standard and undesirable, mainly due to the dual role the properties rdfs:subClassOf, rdf:type, rdfs:domain and rdfs:range play:

> "[These properties] are used both as primitive constructs in the definition of the RDF Schema specification and as specific instances of RDF properties. This dual role makes it possible to view e.g. rdfs:subClassOf as an RDF property just like other predefined or newly introduced RDF properties, but introduces a self-referentiality into the RDF Schema definition, which makes it rather unique when compared to conventional model and meta modeling approaches, and makes the RDF schema specification very difficult to read and formalize."[Nejdl et al., 2000]

The modeling primitives of RDF Schema

RDF Schema defines core classes, core properties and core constraints, that are to be used as modeling primitives for a schema. The RDF Schema specification assigns semantics to a particular set of RDF expressions by describing their intended properties and meaning. We give a brief description of the modeling primitives here:

- **Core classes** are rfds:Resource, rdf:Property, and rdfs:Class. Everything that is described by RDF statements is an instance of the class rdfs:Resource. The class rdf:Property is the class of all properties used to characterize instances of rdfs:Resource. Finally, rfds:Class is used to define concepts in RDFS.
- **Core properties** are rdf:type, rdfs:subClassOf and rdfs:subPropertyOf. The rdf:type relation models instance-of relationships between resources and classes. A resource may be in an instance of more than one class. The rdfs:subClassOf relation models the subsumption hierarchy between classes and is supposed to be transitive. A class may be a subclass of more than one other class, but can not be a subclass of itself, nor is the expressed class hierarchy allowed to be cyclic. Analogous to rdfs:subClassOf, rdfs:subPropertyOf models the subsumption hierarchy between properties.

- **Core constraints** are rdfs:ConstraintResource, rdfs:ConstraintProperty, rdfs:range and rdfs:domain. rdfs:ConstraintResource defines the class of all constraints. rdfs:ConstraintProperty models all properties that are used to define constraints. In the current RDFS specification, it has two instances: rdfs:range and rdfs:domain.

1.6.2 Remarks and observations

The RDF data model only provides binary relations between objects. While this makes for a simple and easy to use model, expressing more complex relations by simulating them with binary relations can quickly become very cumbersome (cf. [van Harmelen and Fensel, 1999]).

In RDF and RDFS, properties are the central modeling primitive. This view was chosen to 'allow anyone to say anything about any existing resource'. However, because of this, and because the data model of RDFS lacks a clear distinction between object- and meta levels, the model is very hard to understand and to use for knowledge modelers.

Another problem with RDF(S) is that the semantics of its primitives are only very loosely defined. The RDF and RDFS specifications only give a natural language description of their intended meaning. In many cases this is ambiguous or incomplete. A direct result of this is that there is no inference model that precisely fits the semantics of the RDF modeling primitives — in fact, there is no inference model at all! This at the very least makes using and extending RDF(S) in a more formal context a non-trivial task.

1.7 A comparison of approaches

In this section, we will compare the approaches presented above on several grounds. This is an excerpt from [van Harmelen and Fensel, 1999].

1.7.1 Supported by Web technology

No matter how nice any knowledge representation language is as proposed by the AI community, the real Web can hardly wait until Netscape and Microsoft decide to support such a language. The order of precedence is the other way round: how well can AI concepts be fitted into the markup languages that are widely supported on the Web, either now or in the foreseeable future.

Unfortunately, this requirement disqualifies a lot of current research aimed at applying AI techniques to the Web. Instead, many of the markup schemes described above, are already (or soon will be) widely supported, with the exceptions of SHOE and OntoBroker (which are syntactic varieties of schemes that are supported).

1.7.2 Avoiding duplication of data

A basic tenet of information modeling is that redundancy almost inevitably leads to inconsistency. It is therefore unfortunate that some of the above markup-schemes enforce a duplication between information for semantic purposes and information for rendering. Of course, no syntax would be able to avoid the possibility of stating redundant information, but we would prefer a syntax that does at least not *necessitate* such redundancy.

The SHOE markup suffers from this drawback. Ontobroker can avoid duplication, but at the cost of using non-standard HTML:

```
This page written by
<A ONTO="TAG[AUTHOR=BODY]">Dieter</A>
```

(where "body" is a reserved word for referring to the text in the tag).

XML is more attractive in this respect, since it uses the same information for both rendering and markup. No duplication of information is required here:

```
<body>
This page written by
<author>Dieter</author>
</body>
```

The same effect can be obtained to a certain extent in HTML, using `SPAN` tags:

```
This page written by
<SPAN CLASS="author">Dieter</SPAN>
```

However, in HTML one is completely committed to the position of text and elements, because this position is important for layout. For example, suppose we want to represent the table in figure 1.1 in HTML with SPAN tags to group personal information:

First name	Jeen	Arjohn
Last name	Broekstra	Kampman
City	Amsterdam	Nijverdal

Table 1.1: a table with personal information

The HTML representation of this would look like this:

```
<table>
    <tr>
        <td>First name</td>
        <td>Jeen</td>
        <td>Arjohn</td>
    </tr>
    <tr>
        <td>Last name</td>
        <td>Broekstra</td>
        <td>Kampman</td>
    </tr>
    <tr>
        <td>City</td>
        <td>Amsterdam</td>
        <td>Nijverdal</td>
    </tr>
</table>
```

In this HTML representation, the relation between concepts (such as the first name and the last name of a person) does not follow the hierarchical nesting of tags. We cannot use nested `SPAN` elements to group information about one person together.

XML does not suffer from this drawback, since in XML content and layout are seperated. In XML, we could represent the table information in any particular serialization we prefer, provided that for rendering purposes we provide a matching stylesheet.

1.7.3 Separation of data and meta data

A strength of RDF is the decoupling of the structure of the document and the structure of the meta-information. RDF makes no strong assumptions on the internal structure of the document that it provides meta data for (unlike, for example, XML, which assumes that the document itself is structured as a labeled tree). As a result, RDF is forced to duplicate information, since it cannot assume that the meta-information is already present in the document itself.

1.7.4 Data models

The data models underlying the various schemes vary greatly:

- HTML only provides a basic **attribute-value** mechanism.
- XML (and HTML with `SPAN` tags) takes **labeled trees** as its data model.
- Ontobroker relies on the very rich **F-logic** data model ([Kifer et al., 1995]).
- RDF's data model is based on **binary relations**, enhanced with a reification mechanism to allow additional meta layers of information. RDFS uses this basic data model to provide a (rather unorthodox) object-oriented type schema on top of RDF.
- SHOE's data model is similar to that of RDFS, but it can express **n-ary relations**. It does, however, not include RDF's reification mechanism.

1.7.5 Ontologies

Ontologies are a specification of the conceptualisation and the corresponding vocabulary used to describe a domain. They can be used to describe the structure of semantics of much more complex objects than common databases and are therefore well-suited for describing heterogeneous, distributed and semistructured information sources such as found on the Web. It is therefore important that any semantic markup-scheme for the Web supports the notion of an explicitly specified ontology.

The HTML approach falls short in this respect: the plain attribute-value data model offers no mechanism for the explicit and seperate specification of a data schema.

HTML-derived approaches such as SHOE and Ontobroker do provide explicit ontologies, albeit in very different ways. In SHOE, ontologies can be defined by information-providers themselves inside their own HTML pages (using a special purpose extension to HTML). Such an ontology contains a class-lattice and possible relations between instances of these classes. Ontobroker ontologies are similar in nature (a class-hierarchy, attributes with domain and range definitions and multiple attribute inheritance), but an essential difference is that Ontobroker relies on a single centrally defined ontology.

The closest thing that XML currently offers for ontological modeling is the Document Type Definition (DTD), which defines the legal lexical nesting of tags in a document. Note that this lexical nesting does not necesssarily correspond with any ontological hierarchy. XML Schema, although still under development, already promises to allow much more expressiveness.

RDF, in combination with RDFS, offers powerful modeling primitives, that can be extended according to need. Basic class hierarchy and relations between classes and objects are expressable in RDFS. However, the model suffers from a lack of distinction between object and meta level, which makes it un-intuitive, and has some other pecularities, such as odd restrictions on the use of domain and range properties: while an indefinite number of domain restrictions can be specified for a property, only one range restriction is allowed. In general, RDF seems to suffer from a lack of formal semantics for its modeling primitives, making interpretation of how to use them properly an error-prone process (see also [Nejdl et al., 2000, Broekstra et al., 2000]).

1.7.6 Conclusions

Perhaps the most surprising outcome of this comparison is that HTML with `SPAN` and `DIV` tags compares relatively well with a more novel approach like XML. Both approaches have a similar data model, and both suffer from the same lack of ontological modeling capacity, though the development of XML Schema is promising in this respect where XML is concerned.

While SHOE and OntoBroker both have their merits as being sound approaches, they suffer from the lack of support for adopting them as a standard.

Despite its drawbacks, RDF(S) seems the most promising approach for meta data annotation sofar, in the sense that it has relatively wide support already, and this support will probably only grow. The problem with the duplication of information is less severe in the On-To-Knowledge context, since the aim is to (semi)automatically generate the meta data by extracting it from the text.

For the purposes of On-To-Knowledge, it seems to us that using RDF(S) as the meta data representation language is the most obvious choice.

Chapter 2

Query languages

2.1 Introduction

Although a representation formalism is an essential building block for a 'Knowledge Web', representing information in a machine-accessible way alone is not enough. Enabling querying and inferencing is of course just as important. This is where query languages come into focus.

Many query languages already exist. The most obvious example is SQL, the standard query language for relational databases. In this chapter, we will explore what properties a query language for semistructured data should have, and what the difference is with existing approaches such as SQL. We will then discuss several proposals for query languages.

2.2 General properties of QLs for semistructured data

In [Abiteboul et al., 1999] a number of requirements for a query language on semistructured data are specified. We will briefly look into some of these requirements here, and specify some additional ones.

Required expressivity

The expressivity required of a query language is not only dependent on the specific format of your data (i.e. is it in XML or something else), but also on what kind of data is represented. For example, when we are representing relational data as semistructured data , we would like our query language to be capable of expressing the operations corresponding to those in standard query languages for relational data, such as SQL. [Abiteboul et al., 1999] points out that this is not a complete requirement, and that many other issues play a role here.

Nevertheless, it seems reasonable for our purposes to require that a query language should be able to express set operations like join and union (in our opinion this is especially important in a Web context, where information is often distributed over several locations), and that it should be able to restructure data.

Semantics

In order to enable query transformation and optimization, precise semantics are needed.

A close fit to the data model

This requirement states that the query language should closely fit the data model underlying the representational format. For example, in the case of XML, where the data model is a labeled tree, the query language should provide means to efficiently traverse the branches of the tree.

Syntactic compactness

While at first sight it may seem desirable to have a query language that is easy to read and interpret by humans, one should remember that in the database world, most SQL queries are machine-generated. Therefore, a simple core language may be more appropriate than a user-friendly language with lots of shortcuts but with unclear semantics.

Compositionality

Simply put, compositionality means that the output of a query is expressed in the same format as the input of a query. This is an essential property for allowing the specification of complex queries as a composite of several more basic queries.

2.2.1 Path expressions

One of the main distinguishing features of query languages for semistructured data is their ability to reach to arbitrary depths in the data graph. To do this, these languages all use the notion of *path expressions*.

A path expression is a simple query, whose result, for a given data graph, is a set of nodes. For example, consider the following bit of XML:

```
<?xml version="1.0"?>
<body>
    This page is written by
    <author>Frank van Harmelen</author>.
    <location>
    His tel.nr. at work is <tel>47731</tel>,
    his number at home is <tel>555722</tel>, and his
    room number is <room>T3.57</room>.
    </location>
</body>
```

The result of the path expression `body.location.tel` would be the set of nodes with the associated values {"47731", "555722"}.

Many useful *regular expressions* can be used in path expressions to facilitate more complex expressions than just specification of the complete path. For example, a regular expression `location|name` specifies either a `location` node or a `name` node. Another very useful pattern is the wildcard, which matches any node label. Using the symbol _ to express this (cf. [Abiteboul et al., 1999]), `body._.tel` matches any path consisting of a `body` node followed by any node, followed by a `tel` node. Also, closure operations, like arbitrary repeats of a regular expression can be used. For example, `body._*.tel` specifies the set of `tel` nodes that occur at arbitrary depth within the `body` node. At another level of abstraction, regular expressions can also be used to express matches on the actual string format of labels. For example the regular expression `body."[aA]uthor"` matches any author node within the body, possibly with the first letter capitalized.

Path expressions, although they are an essential feature of query languages for semistructured data , can only return a subset of nodes in the database. They can not construct new nodes, perform joins, or test values stored in the database. In other words: path expressions are necessary but not sufficient for a good query language on semistructured data .

2.3 Why not just SQL?

For strictly relational data (as opposed to semistructured data), SQL is by far the most widely supported query language, including support for large data-storage, efficient indexing schemes,

query-optimisers, etc. It would therefore be attractive if we could use this robust and widely available technology for our purposes of querying semistructured data .

Unfortunately, this can only be done at the cost of a very large gap between the data-model in the repository (e.g. RDF) and the data-model on which the query-language is based (the relational model).

To exemplify this, let us look at how the scenario would look for an XML implemetation in a relational-database: as a first step, we would have to encode the XML data-model in the relational model. This would be possible by assigning each node in an XML-tree a unique identifier, with each entry in the relational database linking such a node with all its descendants and attributes. The problems start when we want to use this as the basis for querying the XML-structure: each XML-query should be compiled into an SQL-query on the underlying relational tables. Typically, a single XML-query (such as: "return all descendants of a given node") must be compiled into a complicated set of SQL queries. It is not even clear whether a finite set of SQL-queries could be generated for every reasonable XML query.

Although perhaps attractive as a short term solution, we feel that in the long run this is not an appropriate solution. Rather, techniques for large data-storage, indexing schemes, query-optimers, etc. should be provided for the "native" data-model (be it XML or RDF), instead of relying on these techniques for a completely different data model. In fact, work on such techniques for semistructured data is already well underway (e.g. [Abiteboul et al., 1999])

2.4 Querying XML

In this section, we discuss several approaches for a query language for XML, and we will compare them with the general requirements for such a query language as presented in the previous section.

2.4.1 XSL

The **Extensible Stylesheet Language (XSL)** [Adler et al., 2000], is a proposal for a language to express stylesheets for XML documents. It is currently under development at the W3C. XSL is divided in two parts:

- XSLT, a transformation language
- XSL-FO, a set of Formatting Objects

We will focus on the first of these here.

XSLT maps an input data tree to an output data structure. Although its primary role is to allow users to write transformations from XML to HTML describing the presentation of the XML document (i.e. it serves as a stylesheet mechanism), it can also serve in the role of query language.

An XSL program is a set of template rules that are executed in a best-match manner by recursively traversing over the nodes in the input data tree. An example program, which retrieves the names of all animals in the zoo (see figure 2.1):

```
<xsl:template>
    <xsl:apply-templates/>
</xsl:template>
<xsl:template match="/zoo/*/animal/name">
    <result>
        <xsl:value-of/>
    </result>
</xsl:template>
```

```
<zoo>
    <habitat name="Savannah">
        <attendant idref="att1" />
        <animal type="elephant">
            <name> Henri </>
            <gender> male </>
            <favorite_food> peanuts </>
        </animal>
        <animal type="penguin">
            <name> Tux </>
            <gender> male </>
            <favorite_food> fish </>
        </animal>
        <animal type="giraffe">
            <name> Andy </>
            <gender> male </>
            <favorite_food> leafs </>
        </animal>
    </habitat>
    <habitat name="Polar World">
        <attendant idref="att2" />
        <animal type="penguin">
            <name> Frisky </>
            <gender> female </>
            <favorite_food> fish </>
        </animal>
    </habitat>
    <employee id="att1">
        <name> Frank </name>
    </employee>
</zoo>
```

Figure 2.1: An example XML database

Each of the two ⟨**xsl : template**⟩ constructs specifies a template rule. If we look at the second template rule, we see a `match` attribute, which specifies the pattern, and the body of the rule is the template. In the first rule, there is no `match` attribute, which means that the rule is matched by any node in the input data tree.

If we apply this program to the example XML database in figure 2.1, the first template rule is matched by the top level node `zoo`. The ⟨**xsl : apply − templates/**⟩ states that the action `apply-templates` is to be executed on the contents of the `zoo` node. In this manner, the data tree is traversed depth-first, until the search hits the `name` node that matches the more specific second template rule. This rule specifies that when matched, the output should be the value of the current node, enclosed in ⟨**result**⟩ tags.

Applied to the database in figure 2.1, we get the following result:

```
<result> Henri </result>
<ressult> Tux </result>
<result> Andy </result>
<result> Frisky </result>
```

As we can see, XSL has path expressions to allow matching, and uses a recursive descent mechanism to traverse the data model, fitting closely to the XML tree model. However, the things that XSL can not do include binding values to variables, and expressing joins, nor can it express any other boolean operations on sets. Also, the recursive specification of templates seems somehow un-intuitive to make for a good query language. A further disadvantage is that

the context of an XSL program (i.e., the XML document(s) which it should take as its input) is not specified in the program itself, which makes the input mechanism unclear. Concluding, one could say that the stylesheet/transformation background of XSL, while offering much of the same functionality as a query language, makes it less useful for the purpose of querying.

2.4.2 XQL

The **XML Query Language (XQL)** [Robie et al., 1998] is a notation for addressing and filtering the elements and text of XML documents. XQL is a natural extension to the XSL pattern syntax. According to [Robie et al., 1998], it enhances XSL with a.o. boolean operators, filters and indexing capabilities.

XQL extracts data by means of patterns and path expressions, just like we saw with XSL. However, XQL provides far easier syntax and more powerful selection mechanisms.

To sum up the design goals of XQL (from [Robie et al., 1998]):

- XQL strings shall be compact.
- XQL shall be easy to type and read.
- XQL syntax shall be simple for the simple and common cases.
- XQL shall be expressed in strings that can easily be embedded in programs, scripts, and XML or HTML attributes.
- XQL shall be easily parsed.
- XQL shall be expressed in strings that can fit naturally in URLs.
- XQL shall be able to specify any path which may occur in an XML document and specify any set of conditions for the nodes in the path.
- XQL shall be able to uniquely identify any node in an XML document.
- XQL queries may return any number of results, including 0.
- XQL queries are declarative, not procedural. They say what should be found, not how it should be found. This is important because a query optimizer must be free to use indexes or other structures in order to find results efficiently.
- XQL query conditions may be evaluated at any level of a document, and are not expected to navigate from the root of a document.
- XQL queries return results in document order with no repeats of nodes.

An example XQL query, in the context of our zoo:

```
/zoo/habitat
```

This returns all `habitat` elements in our zoo. To find all `habitat` elements anywhere in the zoo (i.e. at arbitrary depth in the tree):

```
/zoo//habitat
```

To find all penguins:

```
//animal[@type = 'penguin']
```

In this last example, we see how XQL differentiates attributes from elements (the prefix) and how one can filter using boolean expressions.

Just like in XSL, in XQL variable binding is still not possible, and hence, no joins can be expressed. The authors claim that nevertheless XQL can be used to query over multiple documents, but this seems to assume that these documents are all available in a single XML repository. However, a mechanism for this is not prescribed by the XQL proposal. Also, XQL does not prescribe an output format. A direct result of this is that XQL can not guarantee compositionality of queries. It also means that XQL does not offer the ability to construct alternative views on data sources. Its usefulness seems to be strictly limited to data retrieval.

2.4.3 XML-QL

XML-QL [Deutsch et al., 1998, Abiteboul et al., 1999] combines XML syntax with query language techniques. It uses path expressions and patterns to extract data from the input XML data, has variables to which this data can be bound and has templates which show how the output XML data is to be reconstructed.

An example of an XML-QL query is:

```
where <habitat type=$T>
        <animal type="penguin">
            <name> $N </name>
            <gender> male </gender>
            <favorite_food> $F </favorite_food>
        </animal>
      </habitat> in "www.a.b.c/zoo.xml"
construct <result>
            <name> $N </name>
            <lives_in> $T </lives_in>
            <food> $F </food>
          </result>
```

As we can see, XML-QL is based on a where/construct syntax, instead of the familiar select/from/where of SQL. The `construct` clause corresponds to `select`, and the `where` combines the `from` and `where` parts of the query, that is, the ranging of variables and some filtering. In the above query, `$T` and `$N` are variables and

```
<habitat type=$T>
   <animal type="penguin">
      <name> $N </name>
      <gender> male </gender>
      <favorite_food> $F </favorite_food>
   </animal>
</habitat>
```

is a pattern. The pattern is matched in all possible ways to the data and the variables are bound to the corresponding values in the matching cases.

In the `construct` part, we see how XML-QL can be used to construct new XML data. The mechanism is simple: we just add a template to the `construct` clause and use bound variables from the `where` clause to fill the template.

So, when we apply this query to the example database of figure 2.1, the result is:

```
<result>
    <name> Tux </name>
    <lives_in> Savannah </lives_in>
    <food> fish </food>
</result>
```

XML-QL can use path expressions, query across multiple data sources, and express joins:

```
where <*.animal> <name> $N1 </name> <*./animal>
     in "zoo.xml",
     <*.animal> <name> $N2 </name> <*./animal>
     in "anotherzoo.xml",
     $N1 = $N2
construct <result> $N1 </result>
```

This query yields all names that animals in *different zoos* share.

To summarize: XML-QL combines relatively easy syntax with powerful query language notions. In fact, XML-QL is relationally complete, that is, its expressivity is on par with SQL when applied to relational data. One particular feature that neither XSL nor XQL share is that it can express joins and thus combine information from multiple data sources.

2.4.4 Conclusion

If we look at the requirements for a query language on semistructured data , as outlined in section 2.2, we see that regarding expressivity XML-QL is the only language of the three presented here that really meets this requirement. For one, it is the only one that can express joins.

The next requirement is an efficient implementation. For both XSL and XQL it is known that engines with relatively good performance exist. A Java implementation for XML-QL does exist[1], but we have yet to see test results for this engine. The authors of XML-QL do point out, however, that since XML-QL is relatively simple, known database techniques for efficient implementation can be easily applied to XML-QL as well.

Another requirement is compositionality. A major problem with both XSL and XQL seems to be that this can not be guaranteed. In both cases, nothing at all is specified or enforced about the specific format of the output. XML-QL seems slightly better in this respect, in that it requires that the output is valid XML, which allows nesting of queries. Lastly, readability of the syntax is an aspect of some importance. XQL seems to offer the most human-understandable syntax, while XSL takes significantly more effort. XML-QL seems to be somewhere inbetween in this respect.

Our preliminary conclusion is that, when judged purely on technical merits, XML-QL seems to offer the best package.

2.5 Lore

Lore [McHugh et al., 1997] is a DBMS designed specifically for managing semistructured data . Its data model is a very simple, self-describing object model called OEM [Papakonstantinou et al., 1995]. In this section, we will take a look at the OEM data model, the Lorel query language and the overall architecture of the Lore system.

2.5.1 OEM

The Object Exchange Model (OEM) is – as the name suggests – a simple Object-Oriented exchange model for data. It serves as the basic data model for various projects at the Stanford University Database Group[2], one of which is Lore.

OEM describes objects as entities having a descriptive label, a type, a value and an object-ID. Each object in OEM has the following structure:

$$\{\texttt{Label}, \texttt{Type}, \texttt{Value}, \texttt{ObjectID}\}$$

where the four fields are:

[1] see http://www.research.att.com/~mff/xmlql

[2] http://www-db.stanford.edu/

- `Label`: a variable length character string describing what the object represents.
- `Type`: the data type of the object's value. Each type is either an atom type (such as integer, string, real, etc.) or the type 'set'. The possible atom types are not fixed and may vary from information source to information source.
- `Value`: A variable length value for the object. The value of an object can itself be a set of objects, thus allowing the creating of aggregated data structures.
- `ObjectID`: A unique variable length identifier for the object, or `null`.

For example (taken from [Papakonstantinou et al., 1995]), we may describe the temperature value 80 degrees Fahrenheit as:

⟨temp-in-Fahrenheit, integer, 80⟩

Since no object ID is specified here, it defaults to `null`. A more complex object, representing a set of two temperatures might look like:

⟨set-of-temps, set, {$cmpnt_1$, $cmpnt_2$}⟩
    $cmpnt_1$ is ⟨temp-in-Fahrenheit, integer, 80⟩
    $cmpnt_2$ is ⟨temp-in-Celcius, integer, 20⟩

Roughly speaking, a database conforming to the OEM data model can be thought of as a graph with complex values as internal nodes, atomic values at leaf nodes, and labeled edges.

The main difference with other OO models is that OEM is much simpler. It supports only object nesting and object identity; features such as inheritance and classes are not supported. The primary reason for this choice is, according to the designers, to facilitate easy integration. Operations for tranforming and merging data will be simpler on a simple data model.

Recently, Lore's data model has been migrated to using an XML-based model (see [Goldman et al., 1999]). This makes the accompanying query language, Lorel, a candidate for an XML Query Language. For now, however, we will focus on the use of OEM as a data model for both repository and query language.

2.5.2 Lorel

Lorel [Abiteboul et al., 1997] is a language designed for querying semistructured data . It is implemented on the top of the Lore DBMS system for semistructured data , developed at Stanford University. The data model that we will assume Lorel uses is OEM, although recently the data model has been migrated to XML. The syntax of Lorel is based on OQL, adopting the familiar `select-from-where` construction. In fact, Lorel can be defined as an extension to OQL, allowing specific constructs tailored for the navigation through semistructured data , such as path expressions. One other aspect of Lorel is that it strongly supports type coercion, which is important when the underlying data is untyped, irregularly typed, or may have missing fields.

Consider the example semistructured data repository in figure 2.2. It encodes knowledge from a restaurant guide. To show how Lorel works, we will present some example queries here.

Example 1: Find the addresses of all restaurants in the 92310 zipcode.

```
select Guide.restaurant.address
where Guide.restaurant.address.zipcode = 92310
```

It is not necessary to know if the zipcode is represented as an integer or a string because Lorel will coerce the values of candidates to a suitable format. It is also important to note that an address that does not contain a zipcode attribute will not generate an error, but will just fail the `where` condition.

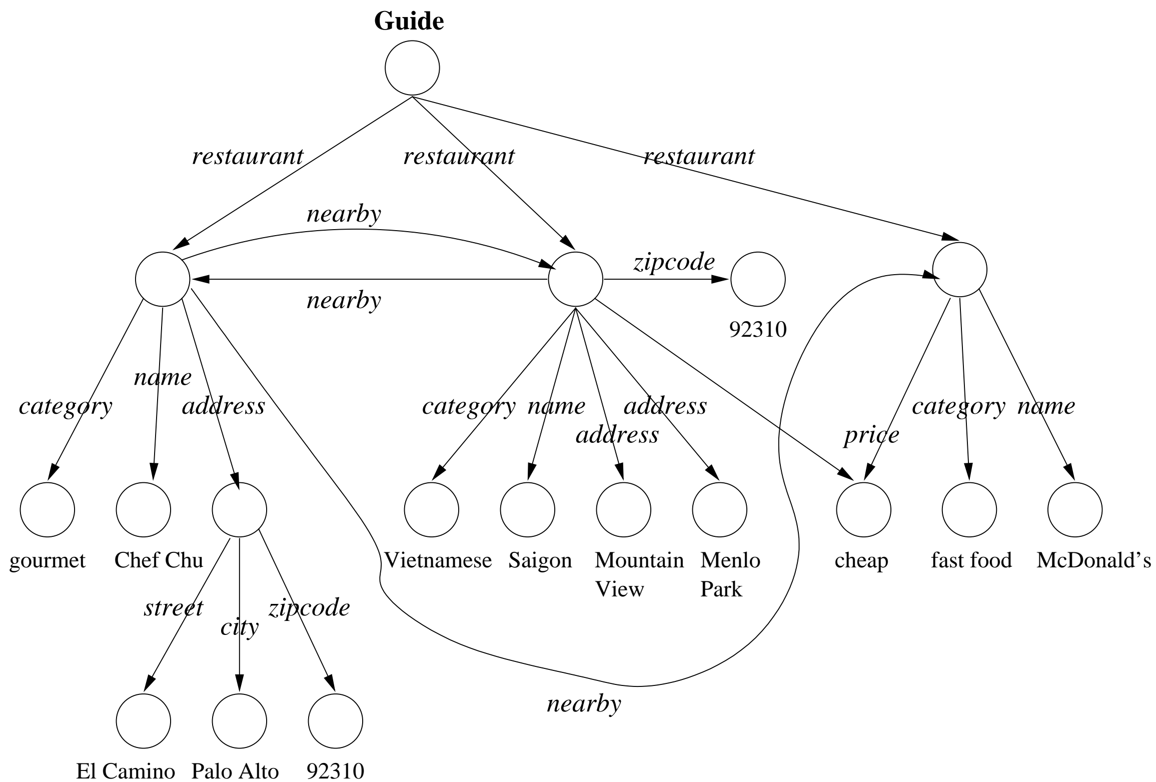

Figure 2.2: an example OEM graph (from [Abiteboul et al., 1997])

The result of this query in the database shown in figure 2.2 will therefore be:

```
address
    street = "El Camino Real"
    city = "Palo Alto"
    zipcode = "92310"
```

Example 2: Find the names and zipcode of all ”cheap” restaurants. Now, we do not assume that the zipcode is a part of the address, it may instead be a direct subobject of the restaurant. Also, we do not know where exactly the string ”cheap” will occur:

```
select Guide.restaurant.name,
       Guide.restaurant(.address)?.zipcode
where Guide.restaurant.% grep "cheap"
```

The ”?” after `.address` means that the address is optional in the path expression. The wildcard ”%” will match any subobject of restaurant, and the comparison operator `grep`[3] will return true if the string ”cheap” appears anywhere in the subobject's value.

In this last example we see again the two typical features of query languages for semistructured data , that 'normal' query languages like SQL or OQL do not have: wildcards and path expressions.

Summarizing: Lorel offers the same expressivity that OQL does, and has added two features specific for querying semistructured data , namely wildcards and path expressions.

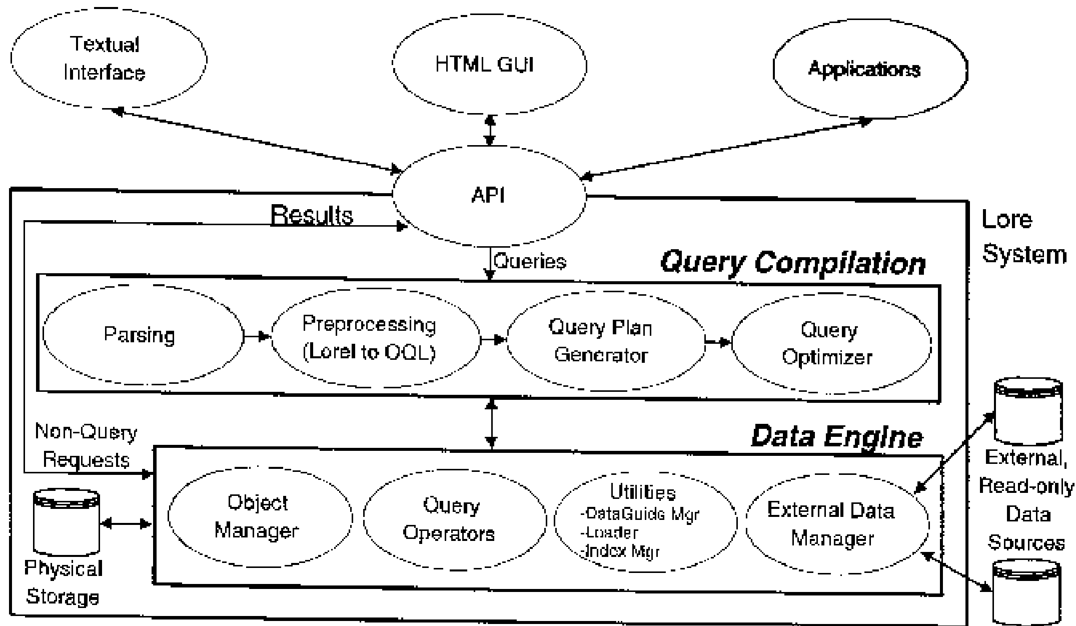

Figure 2.3: The overall architecture of Lore (from [McHugh et al., 1997])

2.5.3 Architecture

The overall architecture is depicted in figure 2.3. We will briefly discuss some of the features most relevant for our purposes, for a detailed explanation we recommend reading [McHugh et al., 1997].

Access to the Lore system is provided though the API. Several interfaces on top of that are provided to enhance interaction, such as an HTML GUI and a textual interface.

The Query Compilation layer consists of a parser, preprocessor, query plan generator and query optimizer. The parser accepts a textual representation of a query, transforms it into a parse tree and passes that tree to the preprocessor.

The preprocessor transforms the Lorel query into an OQL-like query. From this, a query plan is generated and this is then passed on to the optimizer module. The optimized plan is then sent to the data engine layer. We will not go into this layer in detail here.

2.5.4 Remarks and observations

One particularly interesting feature of the Lore system is the compositionality of its query processing. It is very well conceivable that one can adjust Lore to operate with a different query language. Another possible adjustment might be the migration of the data model towards RDF. The query processing unit is OQL based, which is also the basis of choice for a prominent candidate for querying RDF (RQL, see section 2.7.3). Hypothetically, it should be possible to adjust Lore in such a way as to be a storage mechanism for RDF(S) data, and to have an RDF-attuned query language on top. In the worst case, we can still learn valuable lessons regarding implementation and optimization of such a storage mechanism from Lore.

Unfortunately, we have been unable to find any performance analyses of the Lore system, for neither the OEM version nor the XML version. However, the Lore system is available for public

[3]analogous to the Unix command 'grep', which searches for occurences of a specified string in a file

use[4].

2.6 aidministrator's visual rule format

At aidministrator, we have developed a language that was originally intended as a constraint specification language on the structure of documents. This language is part of the WebMaster tool (see [van Harmelen and van der Meer, 1999]), where it is used as a mechanism to classify pages.

To be able to classify pages, we need a way to express properties of Web pages. In WebMaster, this is done by logical rules that capture the user's knowledge on the required content of pages. We assume that the user's knowledge about pages is in terms of the (semantic) markup of pages, and let the rule format follow this assumption. More precisely, the rules will be phrased in terms of the tree structure of the markup in these pages.

In the next section we will look at the logical foundation of the WebMaster rule format. In section 2.6.2, we will look at the visual rule format by means of a few examples.

2.6.1 The logical structure

The following predicate-logical rule format was chosen for the rules in WebMaster (see also [van Harmelen and van der Meer, 1999]):

$$\forall \vec{x}, \vec{y} \Big[\bigwedge_i P_i(x_k, y_l) \Big] \rightarrow \exists \vec{z} \Big[\bigwedge_j Q_j(x_k, z_m) \Big] \tag{2.1}$$

where $\vec{x}, \vec{y}$ and $\vec{z}$ are sets of variables and each of the P_i and Q_j are binary predicates.

Furthermore, when variables represent pages, we can quantify them over types. This is expressed as $x \in T$, where T is the type that x must be a member of.

The binary predicates P_i and Q_j can express one of the following types of relations:

- *Arbitrary nesting of elements:* The predicate $descendant(\langle \texttt{TAG} \rangle\, x\, \langle /\texttt{TAG} \rangle, y)$ is true iff the tagged structure $\langle \texttt{TAG} \rangle\, x\, \langle /\texttt{TAG} \rangle$ occurs somewhere within y.
- *Direct nesting of elements:* The predicate $child(\langle \texttt{TAG} \rangle\, x\, \langle /\texttt{TAG} \rangle, y)$ is true iff the tagged structure $\langle \texttt{TAG} \rangle\, x\, \langle /\texttt{TAG} \rangle$ is one of the *direct* children of y.
- *Simple binary operations:* We will also need simple binary tests on element or attribute contents or on entire pages. Examples are:
  - string-tests on element or attribute content such as string equality, substring, etc.
  - comparisons on ordered types such as integers and calendar dates.
  - tests on (direct and indirect) links between pages.

The difference between descendants and children is essentially that a child is a direct descendant. This comes from the observation that HTML or XML are essentially tree-structures, in which elements are embedded within one another. In fact, WebMaster uses an internal hierarchical representation of web-pages based on W3C's DOM-specification.

As a simple example of a rule, suppose that we want to define homepages as all pages that contain a ⟨PERSON⟩-element. In the class of logical formulae defined above, we can formulate a rule that specifies the constraints that must hold:

$$\forall x \in All_pages : \top \rightarrow \exists z : descendant(\langle \texttt{PERSON} \rangle\, z\, \langle /\texttt{PERSON} \rangle, x)$$

[4] see http://www-db.stanford.edu/lore/home/index.html

It must be stressed that, despite their appearance, these rules are not production rules, but boolean constraint specifications. If the constraint is fulfilled by some instantiation of the rule (i.e. a page), the page becomes a member of the type that is defined by the constraint (in this case, `homepages`).

Obviously, such formulae are cumbersome to define, especially for someone untrained in formal methods, as we have to assume the average website maintainer is. To tackle this problem, a visualization was developed that would be more intuitive to use, and yet provide a 1:1 correspondence with the underlying logical format. This graphical rule notation was deliberately designed to closely reflect the hierarchical structure of web-pages, since the designers expected that the users *will* be familiar with the structure of their own pages (see [van Harmelen and van der Meer, 1999]), and is a semiformal concept map, where a balance has been struck between ease of use, computational support and well-defined semantics. In the next section we see what this visualization looks like, by means of an example rule being specified.

2.6.2 The visual structure

As said before, rules are used in WebMaster to classify pages in types. These types are organized in a hierarchy, rooted by a type called "All Pages". This is the most universal type and always contains every page loaded by the system.

As a first example, we will define the "Homepages" type from the previous section. In the logical format, this rule would be described as follows:

$$\forall x \in All_pages : \top \rightarrow \quad \exists p : [descendant\,(\langle \texttt{PERSON} \rangle\, p\, \langle /\texttt{PERSON} \rangle\, , x)] \tag{2.2}$$

Figure 2.4 shows how this rule would be visualized. The `All pages` node represents a page from the respective type. Attached beneath this node is an element node representing the `PERSON` element. The saw-toothed line is an indication that the element should be nested somewhere within the context of the parent node, i.e. the whole page. The added element is shown in red[5], indicating that it is a demand (part of the right hand side of the implication). All pages that match this rule will become a member of the "Homepages" type, which, as said before, is not a part of the rule itself.

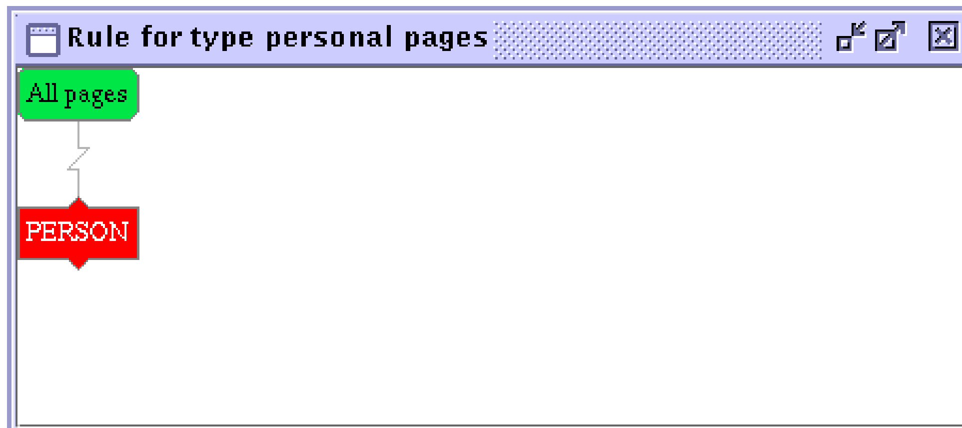

Figure 2.4: The rule specifing type "Homepages"

This is an example of a very simple rule. More complex rules can also be formulated, for example rules where relationships between pages are explored. Imagine the type "Employee Pages", a subtype of "Homepages", on which we want to check whether they have a link to a corresponding resume page, i.e. a resume about the same person (it is exactly this kind of constraint

[5]In a black and white print of this article, the lighter elements are the green ones, and the darker are the red ones

verification for which WebMaster was originally meant). Assuming that we have already defined "Employee Pages" and "Resumes", the logical formula describing this requirement would be:

$$\begin{aligned}\forall x \in employee_pages : \top \rightarrow \quad & \exists y \in resumes\, \exists p, q : [linksto(x, y) \\ & \wedge\, descendant\,(\langle \texttt{NAME} \rangle\, p\, \langle / \texttt{NAME} \rangle\,, x) \\ & \wedge\, descendant\,(\langle \texttt{NAME} \rangle\, q\, \langle / \texttt{NAME} \rangle\,, y) \\ & \wedge\, p = q]\end{aligned} \tag{2.3}$$

This rule defines a so-called *constraint type*, which differs from a regular subtype in that it contains pages that do *not* satisfy the rule populate the type.

This complex logical rule is still easily captured by the visualization used in WebMaster, as can be seen in figure 2.5. It shows the use of additional types against which the pages of the supertype are compared, and the use of binary operators which test for string equality of element content and the existance of a link between two pages.

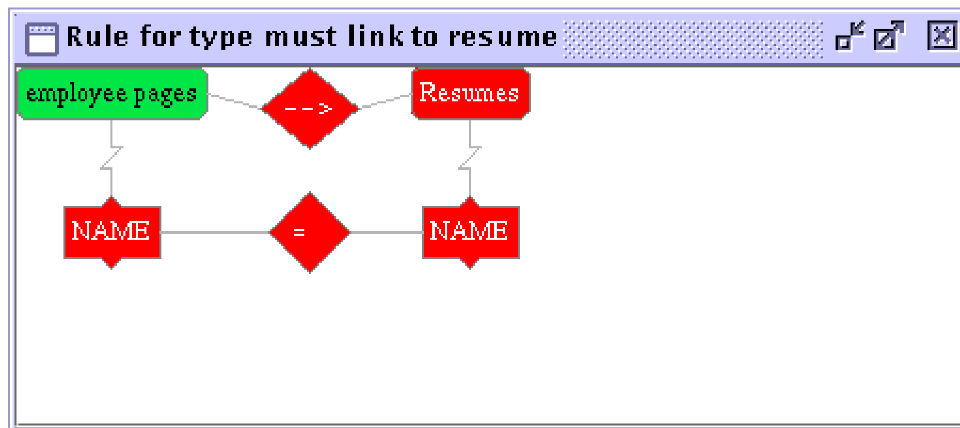

Figure 2.5: The rule specifying the "must link to resumes" constraint.

2.6.3 Comparison with requirements

In section 2.2, we outlined a couple of general requirements for a query language on semistructured data . In this section, we will look at how well the rule format presented here fulfills these requirements.

Required expressivity

The first thing to note with respect to the expressivity is that the rule format is not a full-fledged query language in its present state, because it was never intended as such. The rule format offers powerful features for selection, combination and filtering of information, but a construction part is missing. In other words: The aidministrator rule format cannot be compared to (for example) an XML-QL query (see section 2.4.3), but it is roughly comparable to the `where` clause in such a query.

Another feature which is not available are boolean connectives of predicates other than AND. This is currently under development.

On the other hand, the rule format *is* capable of variable binding and, consequently, able to specify joins, as is shown in the last example. Furthermore, this rule format allows for compositionality of rules, since the output of a rule is used to define a new type.

Readability of syntax

Obviously, much effort has gone into making the graphical syntax easy to use. We believe that this has been quite succesful, though the learning curve is steeper than anticipated. After a short period of 'getting used to', the graphical format and interface prove to be a useful tool for most users.

Efficiency of implementation

The inference engine implemented in the WebMaster tool is quite efficient in itself, for the purposes for which it is used: populating an ontology. Recently, some optimizations have been researched and tested that make it more capable of handling large amounts of data and incremental reasoning (see also [Broekstra, 1999]), which becomes important when the rule format is used to answer queries in an interactive manner.

Conclusion

Although the WebMaster rule format is not a real query language, it features many of the characteristics of one. The merits of the format are mainly in its graphical interface, which allows for easy query manipulation. We think that this is a valuable conclusion to keep in mind for future developments in OnToKnowledge, regardless of the eventual choice of query language for semistructured data .

2.7 Querying RDF

This section (with the exception of 2.7.3) is largely an excerpt from [Karvounarakis, 1999].

Since the initial motivation that led to RDF was the need to represent human-readable but also machine-understandable semantics, it seems obvious that we should use this representation to do something clever with it. The following requirements for an RDF query language can be identified (cf. [Decker et al., 1998]):

- The underlying repository, in which the RDF descriptions are stored, should support the expressive power of the RDF model (assertions), as well as of the RDF Schemata (class and property hierarchies).
- The query language should abstract from the RDF syntax specifications, that is the XML encoding of RDF metadata.
- Such a query language should also provide several facilities - ranging from simple property-value queries, path traversal based on the RDF graph model, to complex Datalog like queries. Several categories of deductions/inferences can be envisaged.
- Subsumption between classes and/or properties not explicitly stated in an RDF Schema using rdfs:subClassOf and rdfs:subPropertyOf.
- Classification of resources: if we know that the domain of a property is of a specific class, we could infer that a resource that appears in a metadata instance, having this property, is actually an instance of this class.

Several approaches for querying RDF have been proposed. These approaches can be generally subdivided into two main categories:

- The SQL/XQL style approach, viewing RDF metadata as a relational or XML database.
- The declarative approach, viewing the Web described by RDF metadata as a knowledge base and, thus, applying knowledge representation and reasoning techniques on RDF metadata.

We will take a look at some examples of these approaches in the following sections.

2.7.1 The SQL/XQL style approach

This first approach, proposed by IBM, is based upon the RDF for XML Java package [Sundaresan, 2000], created by the first. This tool allows one to create RDF objects, as defined in the context of this work, by reading XML encoded RDF metadata (using the syntax employed in the first phase of the creation of RDF). It then provides an API to perform several operations on objects of this class.

Based on this platform, they created a query language [Malhotra and Sundaresan, 1998] in order to be able to query resources, described in these RDF instances. The main constructs of this language are:

- Select, From: as in common SQL for relational databases, these tags are used to define views of the result and declare the domain on which the query is performed. The ”From” tag defines a container, as defined in the RDF Model, which consists of URIs to the metadata descriptions to be queried.
- Condition: This tag is used to describe conditions on the values of properties (similar to the conditions in the ”where” clause of common SQL.

  Example:

```
<rdfquery>
     <From eachResource="http://www/people#PersonsList" >
         <Select properties="Name cooperatesWith">
             <Condition>
                 <equals>
                     <Property name="Name" />
                     <rdf:String>Neel Sundaresan</rdf:String>
                 </equals>
             </Condition>
         </Select>
     </From>
</rdfquery>
```

- Path expressions: This construct can be used in order to navigate across properties in an RDF graph.
- Union, Intersection and Difference: These are used in order to perform the corresponding algebraic set operations on result sets.

  For example, one can issue the following query:

```
<rdfquery>
  <Union>
      <From eachResource="http://www/people/neel">
          <Select>
              <Property name="Publications"/>
          </Select>
      </From>
      <From eachResource="http://www/people/Ashok">
          <Select>
              <Property name="Publications"/>
          </Select>
      </From>
  </Union>
</rdfquery>
```

Union is used in order to join the results of the two queries in a set.

- Group, Order: These constructs allow us to group or sort results, according to the value of a property.
- Quantifiers: The quantifiers forAll and exists are defined in this language, in order to support queries based on the corresponding quantification operators.

Therefore, this QL provides all the constructs that are necessary in order to query specific documents that are described by specific schemata, that is we know in advance which properties each resource has, and what the relations between resources are.

The problem in this approach is that it is not actually what one should expect from an RDF or generally metadata query language. It is actually a transformation of the XQL, in order to be able to query RDF descriptions. It is actually a language for querying data, like XQL, instead of meta data. This should be considered normal, since the authors seem to view RDF just as an XML instance, not as a metadata model that can be encoded using XML syntax for interoperability purposes. This is obvious by the way they model RDF, according to the Document Object Model (DOM) [Apparao et al., 1998], which is used in order to model the structure of XML documents.

Moreover, this approach disregards RDF schema relationships, on which RDF instance documents are based, therefore losing a great deal of the semantics of the descriptions. For example, if there is a description of a ”Person”, as defined in the schema above, and we know that a ”Researcher” cooperatesWith them, we cannot infer that the ”Person” is also a ”Researcher”, as the range constraint in the schema implies. They only consider vocabulary specific inferencing as a future research direction and try to deal with primitive database types that could be supported in the future, which is more of an XML syntax matter than a metadata description problem.

2.7.2 The declarative approach

This second approach is mostly supported by the W3C RDF working group and other related researchers, as well as the founders of RDF itself. It seems that this is the approach of researchers coming from a different community (the knowledge representation community) but it could also be that this approach is more likely to achieve the goals for which RDF was initially proposed. Based on these initial ideas and specifications, several query and inference languages for RDF have been proposed. We will discuss a couple of them here.

A Query and Inferencing Service for RDF

This approach, outlined in [Decker et al., 1998] is practically trying to achieve the aims outlined in section 2.7, by mapping RDF metadata to Frame Logic. By loading the triples extracted from the RDF metadata into an F-logic knowledge base, according to some schema definitions extracted from the included RDF Schemata, they organize the RDF descriptions as a knowledge base. For example, a schema about researchers and papers would be loaded in F-logic by the following F-logic statements (manually in the current implementation):

- Class definitions:

```
Person:: Object.
Researcher:: Person.
Paper:: Object.
```

- Attribute (property) definitions:

```
Person[Name=>> Literal].
Reseacher[Publications=>> Paper;
        CooperatesWith=>> Researcher].
Paper[Year=>> Integer].
```

For instance, in order to perform the query "give me all the resources that have a property Publications", we should say in F-logic:

```
FORALL X, Y <- X[Publication->>Y].
```

Depending on schema definitions, one can perform at least basic inferencing on the RDF descriptions. For example, class hierarchies expressed in F-logic rules, allow inferencing based on subclassing or refining queries by eliminating irrelevant information according to the schema that would otherwise match the query. Constraints in values of properties can also be represented in this way, allowing such a type of validation and inferencing on incomplete information.

For example, according to the schema we know that:

```
Researcher:: Person[Publication=>>Paper].
```

Therefore, according to the semantics of F-logic, we can infer that all persons that have publications are instances of the class "Researcher", even if this fact is not explicitly stated in the RDF descriptions.

Moreover, we can explicitly add inference rules. For example, we know that a Researcher only cooperatesWith another Researcher, so we can add the inference rule:

```
FORALL Person1, Person2
Person1:Researcher[cooperatesWith->>Person2] <-
Person2:Researcher[cooperatesWith->>Person1]
```

This means that if we know that Person2 is a Researcher and cooperatesWith Person1, then we can infer that Person1 is a Researcher and cooperatesWith Person2, even if this is not explicitly stated. Thus, if we then submit a query, requesting all instances of the class Researcher, according to this rule, we are also going to get all resources that a Researcher cooperatesWith.

Another type of inference proposed is automatic query expansion. Moreover, F-logic syntax for the query language seems more user friendly, compared to the RDF encoding, while recursive queries can be performed using the same, F-logic style, query language on the resulting knowledge base. In a similar way, any deductive database (as is F-logic) can be used in the same way.

Finally, F-logic also allows us to perform some kind of queries on the schema. For example, one can request all Researchers that have a property with the value "Christophides". In our case this would mean that either the name or the cooperatesWith property would have that value. This would be encoded in F-logic as:

```
FORALL Res, Prop, T <- Res:Researcher[Prop=>>T] AND
Res[Prop->>"Christophides"]
```

The supposed advantage of the tool SiLRI is that it provides a lightweight and portable implementation of the inference engine described above, including the functionality of F-logic. Therefore, it is probably suitable for mobile code (e.g. intelligent agent applications) implementations.

However, there are some problems with this approach. Firstly, not all RDF expressions can be directly expressed using F-logic. For example, no attempt has been done to describe RDF Containers, as bags etc, and reified statements can not be represented, syntactically and semantically. Moreover, this approach is a common object-oriented one, not regarding properties as first-class objects, as the RDF model suggests. Consequently, no inference can be performed based on property hierarchies.

Furthermore, inference is based upon manually added inference rules; no attempt has been made to define the semantics of RDF itself. No generic inference facilities have been investigated, based on RDF/RDFS semantics, which would apply to any schema instance. What was actually the aim of this work was to be able to load RDF metadata annotations in Ontobroker [Fensel et al., 1998, Fensel et al., 1999] , which uses F-logic as its internal representation language,

since the annotation language proposed by Ontobroker is not widely supported. Moreover, since Ontobroker presumes the existence of a predefined central ontology for each community, the need to be able to load individually created RDF Schema instances is less important to them; all information providers are likely to follow the organization of the ontology. Therefore, specific inference rules are part of the central ontology.

Metalog

This approach, described in [Marchiori and Saarela, 1998] and also related to the W3C RDF working group, is similar at many points to the one described above. It attempts to allow inferencing to be performed on RDF metadata, using knowledge representation and reasoning techniques. However, this approach is closer to the RDF model's property-centric approach, as properties in the RDF model are actually binary relations between resources.

The added value of this proposed system is that it abstracts the query language from the syntax even more. By identifying the correspondence between RDF statements (triples) and predicates in logic, one can view a query language in RDF at a logical layer, allowing any logical relationship to be expressed. What the Metalog approach suggests is an extended RDF Metalog schema, with the addition of logical connectors (and, or, not, implies) and variables to RDF Schema. This way, one can encode inference rules in RDF schema instances.

At this level, one can use the full expressive power of first-order predicate calculus in order to describe the schema of the query language. In order to create a computationally feasible implementation, one can then use a subset of the predicate calculus, as is logic programming. The implementation of Metalog suggests the use of Datalog. This approach has the obvious advantage of allowing the designers to directly address the matter that arises with the trade-off between expressive power and computational efficiency.

According to this approach, apart from the schema definition:

$$Publication(X, Y) \Rightarrow in(X, "Researcher")$$

one can add rules in the schema as:

$$(Publication(X, Y) \wedge Year(Y, Z)) \Rightarrow gotDegreeBefore(X, Z)$$

Then we are able to get all Researchers, who got their degree before a specified year:

$$GotDegreeBefore(X, 1990)$$

Another point that is highlighted in this approach is the need for a user-friendly interface for the query language, actually proposing the use of natural language to express queries. The language proposed maintains all the expressive power of the underlying Metalog schema, and also encapsulates inferencing facilities, together with the ability to express RDF descriptions of any kind.

However, no actual query language is proposed. What Metalog actually suggests is a way to embed inference rules by logical operators in RDF Schemas. Moreover, the architecture of such a system, using Metalog and binary predicate calculus for querying RDF is not obvious. Finally, an implementation of this approach is still to be presented, in order to clarify the ideas of the authors in practice.

RQL

Another approach for a declarative query language, called RQL [Karvounarakis et al., 1999], aims to allow both RDF meta data and schema information to be queried as semistructured graph data.

We will take a closer look at RQL in the next section.

2.7.3 RQL in depth

This section is a summary of [Karvounarakis et al., 1999]. We recommend reading the article for a more detailed description of the language.

RQL adopts the syntax and functional approach of OQL [Cattel and Barry, 1997]. It also features generalized path expressions.

The data model of RQL is a graph model bridging and reconciling the RDF Model with Schema specifications. This model allows to consider RDF instances as semistructured data that can be (partially) interpreted by means of one or more RDF schemas.

RQL is defined by means of a set of **core queries**, a set of **basic filters**, and a way to build new queries through functional composition and iterators.

The core queries provide the means to access information in RDF (meta) data and schemas. For example, the following queries retrieve all the classes or property types in a specific schema:

```
Class
Property
```

`Class` and `Property` are the names of two meta-collections holding the labels of all nodes and edges that can be used in a semistructured RDF (meta)data graph. These basic queries will return the URIs of the classes and property types available in the schema.

Several more core queries are supplied, such as `subClassOf`, `typeOf`. Furthermore, any collection can be accessed by its name, and RQL also considers the names of property types as entry points in the data graph, allowing one to formulate queries such as, for example:

```
creates
```

which will return the set of ordered pairs of resources connected by the `creates` property. Common set operators, such as union and intersection, are also supported.

Finally, RQL supports standard Boolean predicates as $=$, $<$, $>$ and `like` for pattern matching. These operators can be applied to literals or resource URIs.

In order to iterate over collections of RDF (meta)data (e.g., class or property extents, container values, etc.), RQL provides a `select-from-where` filter. The whole data graph can be viewed as a collection of nodes and edges and RQL filters can use *path expressions* to traverse this graph at arbitrary depth.

An example is the simple query:

```
select  X, Y
from    {X}creates{Y}
```

which is equivalent to the basic query `creates`. The `from` clause contains a basic path expression.

The result of an RQL query is serialized using the rdf:Bag construct. Ordered tuples are represented as Sequences. For example, the result of the above query could be (when applied to a database with painters and paintings):

```
<rdf:Bag ID="#result1">
    <rdf:li>
        <rdf:Seq>
            <rdf:li rdf:resource="#picasso"/>
            <rdf:li rdf:resource="#guernica"/>
        </rdf:Seq>
    </rdf:li>
    <rdf:li>
        <!-- etc -->
    </rdf:li>
</rdf:Bag>
```

To build useful filters one can make compositions of path expressions, for example (again applied to our imaginary painters database):

```
select  W
from    {X}first_name{Y}, {Z}paints{W}.has_material{Q}
where   X = Z and Y = "Pablo" and Q = "oil on canvas"
```

which expresses the query *"Find the resources painted by "Pablo" having material "oil on canvas""*.

Furthermore, RQL offers generalized path expressions to cope with incomplete schema knowledge, for example:

*Find the target of properties whose name matches "*name"*.

```
select  Y
from    $P Property {X}$P{Y}
where   $P like "*name"
```

Note that, to distinguish between schema variables and data variables, the schema variables are prefixed with a $.

In [Karvounarakis et al., 1999], a thorough description of the semantics of RQL as well as a detailed description of the data model is given. This concludes the summary of RQL's functionality.

Remarks and observations

If we compare RQL with the requirements expressed in section 2.2, we can come to the following conclusions:

- **expressivity** RQL seems to offer a rich set of core functions and filters, that allow the expression of a great variety of queries.
- **semantics** The semantics of RQL are well defined, see [Karvounarakis et al., 1999].
- **data model** The data model of RQL is a graph model that specifically addresses some peculiarities of the RDF model.

To conclude, we believe that RQL makes an excellent candidate for an RDF query language. However, sofar, RQL only exists on paper. There is no prototype engine (yet), and no tests have been run with it. It is our understanding however, that the research group working on RQL is currently in the process of creating such a test engine.

One additional remark we would like to make is that an interesting course of action might be to try and combine RQL and Lore. Lore uses a query language very similar to RQL (both are OQL-based), although the Lore system is not based on the RDF data model. At the moment, we see two possible ways to go about combining the two:

- migrating Lore to the RDF data model. Whether or not this is possible or even desirable remains to be seen.
- leaving the OEM data model intact, implementing RQL directly on top of the query preprocessing unit. Since in the current system this unit transforms Lorel queries to OQL, it might well be possible to do the same from RQL to OQL. Hover, this requires query expansion in the sense that all knowledge on semantics of RDFS modeling primitives will have to be embedded in the pre-processing unit. Again, whether or not this is desirable and/or practical is an issue for further study.

2.8 A comparison of approaches

In this section we will briefly compare the different approaches with the criteria for query languages that we laid out in section 2.2.

For the ambitious purposes of the On-To-Knowledge project, languages that do not allow the formulation of joins can be dismissed out of hand (XSL and XQL). For XML query-languages, this leaves XML-QL as the only widely-supported candidate. That language comes with a precise semantics, fits closely to the XML data-model, and allows path-expressions and data-transformation. An important lesson from the aidministrator format is the possibility of an easy to understand visual representation for complicated query-expressions.

The landscape for RDF query languages is still somewhat sparsely populated. The best candidate here is RQL, since it is expressive enough (joins, data-construction), has a close fit with the RDF/RDF-Schema data-model, guarantees compositionality, and has a precise semantics.

Chapter 3

Conclusions

In chapter 1 we concluded that RDF was to be preferred as a representation language over XML. The main reason for this is that RDF explicitly commits to specific ontological modelling primitives, whereas XML only provides syntactic constructions without any ontological commitments. The choice for RDF also aligns the On-To-Knowledge project with the W3C initiatives related to the Semantic Web, and with the choices made in the recently started DARPA funded research programme DAML.

This choice for RDF as the data-representation language immediately rules out any of the XML-based query-languages (since we required that the query-language should closely fit to the underlying data-model). Of the RDF query-languages known to us at the time of writing, RQL seems by far the most promising candidate, and fulfills most if not all of our requirements.

One proviso must be made with this choice: RQL firmly commits to the RDF/RDF-Schema data model. This is of course as required, but also implies a limitation: any extensions to the RDF data-model (as is the intended use of RDF-Schema) cannot be directly queried in RQL. Consider for example the case of OIL: it extends the basic RDF/RDF-Schema model with new primitives (e.g. cardinality constraints). Such new elements in the data-model can only be included in a syntactic way in RQL queries, but the RQL query language is not able to process the semantics of such new primitives. Three alternatives exist as a way out of this dilemma:

- define a language which allows for some kind of "semantic plug-in definitions", in order to include the semantics of such schema-extensions into the query language
- implement a new or extended query-language for each extension of the data-model
- write a query-compiler wich compiles queries for the extended language into a set of queries for the original language.

Our choice for RQL as the query-language in the On-To-Knowledge architecture implies that we recommend the third of these options.

While the choice for a storage model is not within the scope of this report, we have already shown that approaches such as Lore suggest that the use of a DBMS specifically tailored for semistructured data is an option worth investigating. Of particular interest to us is whether it is feasible to either migrate Lore to the RDF data model, or to couple RQL to the existing data model in some form. This last option would require the query pre-processing part of Lore to be able to translate RDF primitives into (sets of) queries on the simpler data model of Lore. This approach is similar to the third option we suggested above for the parsing of OIL-specific primitives.

Bibliography

[Abiteboul et al., 1999] Abiteboul, S., Bruneman, P., and Suciu, D. (1999). *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann. ISBN 1-55860-622-X.

[Abiteboul et al., 1997] Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. (1997). The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88.

[Adler et al., 2000] Adler, S., Adler, S., Berglund, A., Caruso, J., Deach, S., Grosso, P., Gutentag, E., Milowski, A., Parnell, S., Richman, J., and Zilles, S. (2000). Extensible Stylesheet Language (XSL). Working draft, World Wide Web Consortium (W3C). See http://www.w3.org/TR/xsl/.

[Apparao et al., 1998] Apparao, V., Byrne, S., Champion, M., Isaacs, S., Jacobs, I., Hors, A. L., Nicol, G., Robie, J., Sutor, R., Wilson, C., and Wood, L. (1998). Document Object Model (DOM) Level 1 Specification. Recommendation, World Wide Web Consortium (W3C). See http://www.w3.org/TR/REC-DOM-Level-1/.

[Bray et al., 1998] Bray, T., Paoli, J., and Sperberg-McQueen, C. (1998). Extensible Markup Language (XML) 1.0. Recommendation, World Wide Web Consortium (W3C). See http://www.w3.org/TR/1998/REC-xml-19980210.

[Brickley and Guha, 2000] Brickley, D. and Guha, R. (2000). Resource Description Framework (RDF) Schema Specification. Candidate recommendation, World Wide Web Consortium (W3C). See http://www.w3.org/TR/rdf-schema/.

[Broekstra, 1999] Broekstra, J. (1999). Efficient strategies for reasoning about Web sites. Master's thesis, Faculty of Sciences, division of Computer Science and Mathematics, Vrije Universiteit.

[Broekstra et al., 2000] Broekstra, J., Klein, M., Fensel, D., Decker, S., and Horrocks, I. (2000). OIL: a case-study in extending RDF-Schema. To appear.

[Byron and Malhotra, 2000] Byron, P. V. and Malhotra, A. (2000). XML Schema Part 2: Datatypes. Working draft, World Wide Web Consortium (W3C). See http://www.w3.org/TR/xmlschema-2/.

[Cattel and Barry, 1997] Cattel, R. and Barry, D. (1997). *The Object Database Standard ODMG 2.0*. Morgan Kaufmann.

[Decker et al., 1998] Decker, S., Brickley, D., Saarela, J., and Angele, J. (1998). A Query and Inference Service for RDF. See http://purl.org/net/rdf/papers/QL98-queryservice.

[Deutsch et al., 1998] Deutsch, A., Fernandez, M., Florescue, D., Levy, A., and Suciu, D. (1998). XML-QL: A Query Language for XML. Position paper ql'98 workshop, World Wide Web Consortium (W3C). See http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/.

[Fallside, 2000] Fallside, D. A. (2000). XML Schema Part 0: Primer. Working draft, World Wide Web Consortium (W3C). See http://www.w3.org/TR/xmlschema-0/.

[Fensel et al., 1999] Fensel, D., Angele, J., Decker, S., Erdmann, M., Schnurr, H.-P., Staab, S., Studer, R., and Witt, A. (1999). On2broker: Semantic-Based Access to Information Sources at the WWW. In *Proceedings of the World Conference on the WWW and Internet (WebNet 99)*.

[Fensel et al., 1998] Fensel, D., Decker, S., Erdmann, M., and Studer, R. (1998). Ontobroker: Or How to Enable Intelligent Access to the WWW. In *Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW98)*.

[Goldman et al., 1999] Goldman, R., McHugh, J., and Widom, J. (1999). From semistructured data to xml: Migrating the lore data model and query language. In *Proceedings of the 2nd International Workshop on the Web and Databases (WebDB '99)*.

[Heflin et al., 1999] Heflin, J., Hendler, J., and Lue, S. (1999). SHOE: A knowledge representation language for internet applications. Technical report cs-tr-4078 (umiacs tr-99-71), Institute for Advanced Computer Studies, University of Maryland, College Park.

[Heflin et al., 1998] Heflin, J., Hendler, J., and Luke, S. (1998). Reading Between the Lines: Using SHOE to Discover Implicit Knowledge from the Web. In *AAAI-98 Workshop on AI and Information Integration*.

[Horrocks et al., 2000] Horrocks, I., Fensel, D., Broekstra, J., Decker, S., Erdmann, M., Goble, C., van Harmelen, F., Klein, M., Staab, S., Studer, R., and Motta, E. (2000). The Ontology Inference Layer OIL. Technical report, Vrije Universiteit. See http://www.ontoknowledge.org/oil/.

[Hoschka, 1998] Hoschka, P. (1998). Synchronized Multimedia Integration Language (SMIL) 1.0 Specification. Recommendation, World Wide Web Consortium (W3C). See http://www.w3.org/TR/REC-smil.

[ISO8879, 1986] ISO8879 (1986). Information processing – Text and Office Systems – Standard Generalized Markup Language (SGML). Technical report, International Organization for Standardization.

[Karvounarakis, 1999] Karvounarakis, G. (1999). RDF Query Languages: A state-of-the-art. See http://www.ics.forth.gr/proj/isst/RDF/QL/rdfql.html.

[Karvounarakis et al., 1999] Karvounarakis, G., Christophides, V., and Plexoussakis, D. (1999). Querying Semistructured (Meta)Data and Schemas on the Web: The case of RDF and RDFS.

[Kifer et al., 1995] Kifer, M., Lausen, G., and Wu, J. (1995). Logical foundations of object-oriented and frame-based languages. *Journal of the ACM, 42*.

[Klein et al., 2000] Klein, M., Fensel, D., van Harmelen, F., and Horrocks, I. (2000). The relation between ontologies and schema-languages: translating OIL-specifications into XML-Schema. To appear.

[Lasilla and Swick, 1999] Lasilla, O. and Swick, R. R. (1999). Resource Description Framework (RDF) Model and Syntax Specification. Recommendation, World Wide Web Consortium (W3C). See http://www.w3.org/TR/REC-rdf-syntax/.

[Malhotra and Sundaresan, 1998] Malhotra, A. and Sundaresan, N. (1998). RDF Query Specification. See http://www.w3.org/TandS/QL/QL98/pp/rdfquery.html.

[Marchiori and Saarela, 1998] Marchiori, M. and Saarela, J. (1998). Query + Metadata + Logic = Metalog. See http://www.w3.org/TandS/QL/QL98/pp/metalog.

[McHugh et al., 1997] McHugh, J., Abiteboul, S., Goldman, R., Quass, D., and Widom, J. (1997). Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66.

[Nejdl et al., 2000] Nejdl, W., Wolpers, M., and Capelle, C. (2000). The RDF Schema Specification Revisited. In *Modelle und Modellierungssprachen in Informatik und Wirtenschaftsinformatik*. Workshop Modellierung 2000.

[Papakonstantinou et al., 1995] Papakonstantinou, Y., Garcia-Molina, H., and Widom, J. (1995). Object exchange across heterogeneous information sources. In *Proceedings of Data Engineering (ICDE), Taipei, Taiwan*.

[Raggett et al., 1999] Raggett, D., Hors, A. L., and Jacobs, I. (1999). HTML 4.01 Specification. Recommendation, World Wide Web Consortium (W3C). See http://www.w3.org/TR/html4/.

[Robie et al., 1998] Robie, J., Lapp, J., and Schach, D. (1998). XML Query Language (XQL). Position paper ql'98 workshop, World Wide Web Consortium (W3C). See http://www.w3.org/TandS/QL/QL98/pp/xql.html.

[Sundaresan, 2000] Sundaresan, N. (2000). RDF for XML. See http://www.alphaworks.ibm.com.

[Thompson et al., 2000] Thompson, H. S., Beech, D., Maloney, M., and Mendelsohn, N. (2000). XML Schema Part 1: Structures. Working draft, World Wide Web Consortium (W3C). See http://www.w3.org/TR/xmlschema-1/.

[van Harmelen and Fensel, 1999] van Harmelen, F. and Fensel, D. (1999). Practical Knowledge Representation for the Web. In Fensel, D., editor, *Proceedings of the IJCAI'99 Workshop on Intelligent Information Integration*.

[van Harmelen and van der Meer, 1999] van Harmelen, F. and van der Meer, J. (1999). WebMaster: Knowledge-based verification of web-pages. In Imam, I., Kodratoff, Y., and Ali, M., editors, *Twelfth International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems IEA/AIE'99*, number 1611 in Lecture Notes in Artificial Intelligence, pages 256–265. Springer Verlag.