WISDOM

Programma di ricerca (cofinanziato dal MIUR, esercizio 2004)
Ricerca Intelligente su Web basata su Ontologie di Dominio
Web Intelligent Search based on DOMain ontologies

Critical analysis of query languages and ontology-based query rewriting techniques

M. Golfarelli, F. Mandreoli, R. Martoglia, A. Proli, S. Rizzi, P. Tiberio

D3.R1 **17 giugno 2005**

Sommario

In order to achieve the ultimate goal of the WISDOM project, i.e. to develop intelligent techniques and tools, based on domain ontologies, performing effective and efficient information search on the WEB, suitable solutions have to be employed in order to express and process queries in the heterogeneous/distributed ontologies scenario and to present the obtained results to the user. To this end, in this report we will perform a critical analysis of the main *query languages* for XML and for the Semantic Web, which may be used as a good basis for supporting the querying needs of the Wisdom project. Then, we will provide an analysis of the approximate query answering issue, which is central in Wisdom, examining available structural *query rewriting* techniques which were proposed and developed for searching in heterogeneous XML document bases or in distributed peer data management systems involving ontology data. Finally we will provide an evaluation of the currently available tools involving the *knowledge presentation* aspect.

Tema	Tema 3: Elaborazione di interrogazioni
Codice	D3.R1
Data	17 giugno 2005
Tipo di prodotto	Rapporto tecnico
Numero di pagine	34
Unità responsabile	MO
Unità coinvolte	MO, TN, BO
Autori	M. Golfarelli, F. Mandreoli, R. Martoglia, A. Proli, S. Rizzi, P. Tiberio
Autore da contattare	Federica Mandreoli, mandreoli.federica@unimo.it

Critical analysis of query languages and ontology-based query rewriting techniques

M. Golfarelli, F. Mandreoli, R. Martoglia, A. Proli, S. Rizzi, P. Tiberio

17 giugno 2005

Abstract

In order to achieve the ultimate goal of the WISDOM project, i.e. to develop intelligent techniques and tools, based on domain ontologies, performing effective and efficient information search on the WEB, suitable solutions have to be employed in order to express and process queries in the heterogeneous/distributed ontologies scenario and to present the obtained results to the user. To this end, in this report we will perform a critical analysis of the main *query languages* for XML and for the Semantic Web, which may be used as a good basis for supporting the querying needs of the Wisdom project. Then, we will provide an analysis of the approximate query answering issue, which is central in Wisdom, examining available structural *query rewriting* techniques which were proposed and developed for searching in heterogeneous XML document bases or in distributed peer data management systems involving ontology data. Finally we will provide an evaluation of the currently available tools involving the *knowledge presentation* aspect.

1 Introduction

In recent years, the constant integration and enhancements in computational resources and telecommunications, along with the considerable drop in digitizing costs, have fostered an enormous increase of data and services available on the Web. In such a sea of electronic information, the user can easily get lost in her/his struggle to find the information (s)he requires. This problem requires the development of novel tools for the integration, the localization and the customizable fruition of informative resources and, in particular, of systems that overcome the "information overloading" problem of traditional search engines.

Indeed, the goal of the WISDOM project is to develop intelligent techniques and tools, based on domain ontologies, to perform effective and efficient information search on the WEB. In order to achieve this goal suitable solutions have to be employed in order to express and process queries in the heterogeneous/distributed ontologies scenario and present their results. Specifically, the following points are fundamental:

- to adopt / devise an expressive enough query language, carefully analysing the features of the ones which are available for querying XML Data and of the most promising Semantic Web Languages;
- to define effective techniques for automatic query rewriting which, by exploiting the information on the semantics of the single concepts described in the reference ontologies and the context where they are placed, reformulate the query against the other ontologies in a form that closely matched the original one;
- to develop methods that allow the result to be interactively navigated, according to the abstraction levels offered by ontologies.

In order to facilitate these precise goals, in this report we will first (Section 2) perform a critical analysis of the main *query languages* for XML, in particular the XQuery one, and the ones for the Semantic Web, which may be used as a good basis for supporting the querying needs of the Wisdom project. Then, since the large power and flexibility of structural query languages give rise to several issues in effectively solving the query answering problem, in Section 3 we will first give an overview of the problem of structural query matching evaluation, then we will deepen the analysis of approximate query answering, which is a central issue in the Wisdom scenario. In particular, we will analyze available structural *query rewriting* techniques which were proposed and developed for searching in heterogeneous XML document bases or in distributed peer data management systems involving ontology data. Finally, in Section 4 we will evaluate the currently available tools involving the *knowledge presentation* aspect. Indeed, research in knowledge presentation has recently led to the development of several visual languages, user interaction paradigms and corresponding tools whose features have to be analyzed in order to fully achieve the Wisdom goals also from the points of view of information representation and user interaction.

2 Query Languages for XML and the Semantic Web

We present in this section a short survey of the main query languages for XML and the Semantic Web. These languages, and in particular the XQuery one, may be used as a good basis for supporting the querying needs of the Wisdom project. In particular, we present the XQuery language (subsection 2.1), which is the major candidate standard for querying XML Data, and the most promising Semantic Web Languages, namely RQL (subsection 2.2), RDQL (subsection 2.3), OWL-QL (subsection 2.4) and SWQL (subsection 2.5).

2.1 XQuery

Extensible Markup Language (XML) [14] has quickly become the *de facto* standard for data exchange and for heterogeneous data representation over the Internet. Consider, for instance, the Digital Libraries context. In recent years, the constant integration and enhancements in computational resources and telecommunications, along with the considerable drop in digitizing costs, have fostered development of such systems, which are able to electronically store, access and diffuse via the Web a large number of digital documents and multimedia data. Digital Libraries more and more often collect documents coming from different sources, usually available on the web. Such documents are heterogeneous for what concerns their representation structures their but are related concerning the contents they deal with. XML is considered the format of choice for the exchange of information in this field and, more generally, among most applications on the Internet, due mainly to its flexibility for representing merely any information as well as metadata for any kind of digital objects. Along with XML, languages for querying XML documents are becoming more and more popular. In fact, given those premises, it is natural that queries among applications should be expressed as queries against data in XML format. Among the several standard query languages proposed in recent years, XQuery [29, 18], currently a W3C working draft, is one of the most promising forthcoming standards.

XQuery, which has been influenced by most of the previous XML query languages, is a language for querying XML data. XQuery derives from Quilt, which, in turn, was inspired and took several characteristics from a number of other XML query languages, such as XQL and XML-QL, and even from relational data query languages, such as SQL and OQL. One of the starting points for the design of XQuery was the fact that XML data is very different from relational data: While the latter tend to have a regular and "flat" structure, the former often contain many levels of *nested* elements. Moreover, relational data are generally *unordered* and are usually "*dense*" (nearly every field of a relation has a value), while XML documents

have an intrinsic *order* and are often "sparse", representing missing information simply by the absence of an element. XQuery is designed in order to exploit such peculiar features in retrieving and interpreting information from diverse XML data sources. Its design makes it applicable in structure-based information retrieval found in *human-readable XML documents*, where it can be employed to retrieve individual documents, to provide dynamic indexes, to perform context-sensitive searching, and to generate new documents. Moreover, XQuery may be used in *data-oriented documents*, in order to query (virtual) XML representations of databases, to transform data into new XML representations, and to integrate data from multiple heterogeneous data sources.

XQuery is very flexible and provides two different syntaxes: One that is optimized for human writing and understanding, which is the most used and is the one which we will briefly discuss, and one that is expressed in XML. The language is defined in terms of a *data model* based on heterogeneous sequences of nodes and atomic values, and is functional, consisting of several types of *expressions* (path expressions, element constructors, function calls, arithmetic and logical expressions, conditional expressions, quantified expressions, and so on). The different kinds of expressions can be combined, resulting in even more powerful expressions. A *query* provides a mapping from one instance of the data model to another instance of the data model. A library of predefined *functions* [26] is provided, and users are also allowed to define functions of their own.

The XQuery data model [27] is based on that of XPath and treats each XML document as a tree of nodes. A *sequence* is an ordered collection of zero or more items. An *item* may be a *node* (element, attribute, text, document, comment, and so on) or an *atomic value* such as a string, an integer, a date or any of the built-in data types. A node may have other nodes as children, thus forming one or more node hierarchies. Among all the nodes in a hierarchy there is a total ordering called *document order*, in which each node appears before its children. Input XML documents can be transformed into the query data model by a process called *schema validation*, which parses the document, validates it against a particular schema, and represents it as a hierarchy of nodes and atomic values, labeled with type information derived from the schema. If an input document does not have a schema, it is validated against a default schema. The result of a query may be transformed from the query data model back into an XML representation by a process called *serialization*.

XQuery has several kinds of expressions. The simplest ones are *literals*, which represent atomic values, and *variables*, names that begin with a dollar sign and that may be bound to a value or used to represent a value. Among the most important expressions are *path expressions*, which are based on the syntax of XPath. A path expression consists of a series of steps, separated by the slash character ("/"). Each step is evaluated in the context of a particular node, called the *context node*. The result of each step is a sequence of nodes and the value of the path expression is the node sequence that results from the last step in the path. As a path expression is evaluated, the nodes selected by each step serve in turn as context nodes for the following step. For example, the path expression `cd/singer` selects the `singer` elements, which are children of the `cd` elements. Further, *predicates* can be used to filter the elements retrieved in each step, e.g. `cd[name=''Ultra'']/singer`, and different *axes* can be employed in order to decide the direction of the data tree traversal (the default direction is from the parent node to its child).

Even more powerful than path expressions are *FLWOR* (pronounced "flower") expressions, which are the analogue of the SELECT-FROM-WHERE construction in SQL. FLWOR expressions form the skeleton of a standard XQuery expression. A *for* clause generates an ordered list of bindings, introducing a variable, *let* associates further bindings, *where* filters the retrieved element list (which is sorted by *order by*) and *return* constructs the final result. In its most general form, a FLWOR expression may have multiple for clauses, multiple optional let clauses, an optional where clause, an optional order by clause, and a return clause. FLWOR expressions are very powerful; they can select existing nodes, as well as construct new elements and attributes

and specify their contents and relationships, by means of particular kinds of expressions, the *element constructors*. Figure 1 shows an example of a FLWOR expression, selecting the names of the music stores which have in their stock a cd featuring a particular singer:

```
for $x in /musicStore
where $x/storage/cd/singer = "Elisa"
return $x/name
```

Figure 1: A FLWOR expression in XQuery

By means of path, FLWOR and many other expressions offered by the specifications, such as conditional and quantified ones, XQuery is able to reach a particularly significant level of flexibility in querying XML data. Still, it is able to maintain a very high level of interoperability among different systems: For instance, any user interface that generates a query in XQuery is able to access any system, such as a digital library one, supporting such a language. Among the other strengths of XQuery are the following: It is completely general (other query languages are often too specific) and therefore broadly applicable, easily understood, and fully integrated with the other XML languages and standards, such as XML Schema and XPath.

2.2 RQL

RQL [40] is a query language adapting the functionality of semi-structured and XML query languages to the peculiarities of RDF. This functionality is extended in order to uniformly query both RDF descriptions and schemas. RQL is a typed language that follows a functional a-la OQL [17] approach and relies on a formal graph model that permits the interpretation of superimposed resource descriptions created using one or more RDF schemas. RQL is defined by a set of basic queries and iterators, which form the building blocks for defining new queries through functional composition. RQL supports generalized path expressions featuring variables on labels for both nodes (i.e. classes) and edges (i.e. properties). RQL queries can be classified in the following categories:

- **Basic (Core) Queries.** The core RQL queries essentially provide the means to access RDF description bases with minimal knowledge of the employed schemas, thus allowing the implementation of a simple browsing interface for RDF description bases. The basic RQL queries allow retrieving the contents of any collection with RDF data or schema information. RQL provides a select-from-where filter to iterate over these collections and introduce variables. Path expressions can be used in RQL filters to traverse RDF graphs at arbitrary depths. The basic RQL query functionality includes:
  - **Accessing Classes and Properties.** Access to classes and properties using their names is provided.
  - **Class Hierarchy Traversal.** Class hierarchies may be traversed using either the `subClassOf` function, which returns the transitive subclasses of a class, or the `subClassOf^` function, which returns the direct subclasses of a class.
  - **Property Hierarchy Traversal.** Property hierarchies may be traversed using either the `subPropertyOf` function, which returns the transitive subproperties of a property, or the `subPropertyOf^` function, which returns the direct subproperties of a property.
  - **Property Definition.** Property definition features may be accessed using appropriate functions: The `domain` function returns the classes in the domain of which the property in question is defined, while the `range` function returns the range of the property (e.g. integers between 1 and 10).

- **Schema Queries.** Simple schema querying capabilities, using appropriate metaclasses are provided at the core level. The default metaclasses provided are the `Class` and `Property` metaclasses.
- **Set Operators.** The `union`, `intersection` and `minus` functions provide the functionality of the basic set operators.
- **Sequences.** Access to sequence members is provided through the `[]` operator.
- **Boolean Operators.** The basic boolean operators >, <, `=` and `like` are provided.
- **Aggregate Functions.** The common aggregation functions `min`, `max`, `avg`, `sum` and `count` are provided.

- **Schema Queries.** The schema querying facilities allow querying RDF schemas, regardless of any underlying instances. RQL extends the notion of generalized path expressions to entire class (or property) inheritance paths, in order to implement schema browsing or filtering using appropriate conditions. The {} notation is used to introduce appropriate schema or data variables. Class variables are prefixed by `$` and range by default over the extent of the RDF metaclass `Class`, which contains all the classes. Using the above facilities the natural language query "Which classes can appear as domain and range of the `name` property?" is expressed in RQL as shown in Figure 2.

```
select $C1 $C2 from {$C1}name{$C2}
```

Figure 2: The query "Which classes can appear as domain and range of the `name` property?" in RQL

Property variables are prefixed by `@` and range by default over the extent of the built n `Property` metaclass, containing all the properties. Thus, the natural language query "Find all properties that are applicable on class `Person`, as well as the property ranges" is expressed in RQL as shown in Figure 3.

```
select @P, range{@P}
from {$C}@P
where $C=Person
```

Figure 3: The query "Find all properties that are applicable on class `Person`, as well as the property ranges" in RQL

The `.` (dot) notation is used to introduce join conditions between the left and the right part of a path expression, depending on the type of each path component.

- **Data Queries.** They allow using generalized path expressions in order to navigate and filter RDF description bases without taking into account the domain and range restrictions. As an example, the natural language query "Find all the Library resources published after 2000" is expressed in RQL as shown in Figure 4.

```
select X, Y
from Library{X}.published{Y}
where Y>=2001-01-01
```

Figure 4: The query "Find all the Library resources published after 2000" in RQL

- **Combinations of Schema and Data Queries.** Queries combining both schema and data querying allow to start querying resources according to one schema, while discovering in the sequel how the same resources are described using other schemas. As an example, the natural language query "Find the description – under the form of triples – of resources excluding properties related to the class `Person`" is expressed in RQL as shown in Figure 5.

```
((select X, @P, Y from {X}@P{Y})
union
(select X, type, $W from $W{X}))
minus
((select X, @P, Y from {X;Person}@P{Y})
union
(select X, type, Person from Person{X}))
```

Figure 5: The query "Find the description – under the form of triples – of resources excluding properties related to the class `Person`" in RQL

RQL has been used as the query interpreter of the RSSDB [9] persistent RDF Store, which is implemented on top of the PostgreSQL Object Relational DBMS (ORDBMS). In addition, a full-fledged view definition language, the RVL [44] language, has been developed based on RQL. An RDF Triggering Language, namely RDFTL [51], has also been developed on top of RQL.

2.3 RDQL

The RDQL (RDF Data Query Language) [56, 55] is an SQL-like language developed in the HP Labs and has been used in several RDF [34] systems for extracting information from RDF metadata repositories. RDQL is "data-oriented", as there is no inference done: with RDQL only the information held in the RDF models is queried. As RDF models are essentially graphs, often expressed as a set of triples, an RDQL query consists of:

- A graph pattern, expressed as a list of triple patterns. Each triple pattern is comprised of named variables and RDF values (URIs and literals).
- An optional set of constraints on the values of the named variables.
- An optional list of the variables required in the answer set.

An RDQL query treats an RDF graph purely as data. If the implementation of the graph provides inferencing to appear as "virtual triples" (i.e. triples that appear in the graph but are not in the ground facts) then those triples will be included as possible matches in triple patterns. RDQL makes no distinction between inferred triples and ground triples.

An example of an RDQL query is given in Figure 6. In the query of Figure 6, the query result is comprised of the resources found in `http://example.org/someWebPage`, with age=24. Age is structured according to the definition found in `http://example.org/peopleInfo#`, represented by the "info" namespace.

2.4 OWL-QL

The OWL Query Language (OWL-QL) [31] is a prototypical formal language and protocol to be used in query-answering dialogues between a querying agent (client) and an answering agent (server). The underlying knowledge should be represented in the Ontology Web Language

```
SELECT ?resource
FROM http://example.org/someWebPage
WHERE (?resource info:age ?age)
AND ?age = 24
USING info FOR http://example.org/peopleInfo#
```

Figure 6: An RDQL Query

(OWL) [46]. OWL-QL is easily adaptable to other declarative formal logic representation languages, including first-order logic languages such as KIF and the earlier W3C languages, RDF, RDF-S, and DAML+OIL.

In a query-answering dialogue, answers are delivered by the server in *answer bundles*. The client may specify, setting an *answer bundle size bound* for each query, the maximum number of answers in each bundle. The server is then required to deliver an answer bundle containing at most the number of query answers given by the answer bundle size bound. In addition, an answer bundle must contain either a *process handle* or one or more character strings called *termination tokens*. The presence of a process handle in an answer bundle represents a commitment by the server to deliver another answer bundle if more answers to the query are requested by the client, while the presence of a termination token in an answer bundle indicates that the server will not deliver any more answers to the query. OWL-QL specifies three termination tokens:

- **"End"**. It indicates that the server is unable to deliver any more answers and is used to terminate the process of responding to a query.
- **"None"**. It expresses a server assertion that no other answers are possible.
- **"Rejected"**. It is used by the server to indicate that the query is outside its scope for some reason, e.g., it is ill-formed or it uses a subset of the language the server is unable to process.

A client may request additional answers to a query by sending the server a *server continuation* containing the process handle provided by the server in the previously produced answer bundle. Upon receiving a server continuation, the server is expected to respond by sending to the requesting client another answer bundle. The dialogue is terminated when the client sends a *server termination* containing the process handle provided by the server in the previously produced answer bundle. The structure of an OWL-QL query-answering dialogue is illustrated in Figure 7.

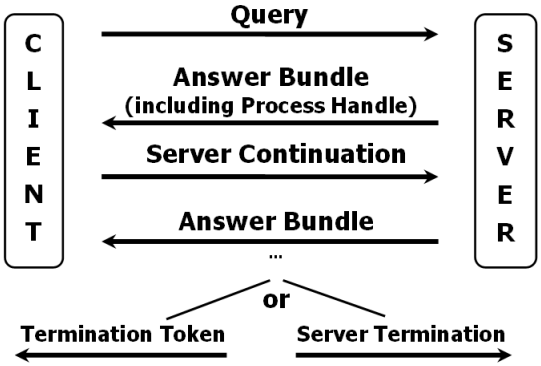

Figure 7: OWL-QL Query-Answering Dialogue

An OWL-QL query is an object containing a query pattern[1] that specifies a collection of OWL sentences in which some URIrefs are considered to be variables. The variable notion does not exist in OWL, thus an OWL-QL query pattern is simply an OWL knowledge base, and a query specifies which URI references in its query pattern are to be considered to be variables. For example, a client could ask "Who owns a black car?" with a query having the query pattern shown in Figure 8.

```
Query:  (‘‘Who owns a black car?’’)
 Query Pattern:  {(owns ?p ?c) (type ?c Car) (has-color ?c Black)}
 Must-Bind Variables List:  (?p)
 May-Bind Variables List:  ()
 Don’t-Bind Variables List:  ()
Answer:  (‘‘Matt owns a black car?’’)
 Answer Pattern Instance:  {(owns Matt ‘‘a black car’’)}
```

Figure 8: A sample Query-Answer in OWL-QL

OWL-QL enables the client to designate some of the query variables for which answers will be accepted with or without bindings in order to support existentially quantified answers. Thus, each variable occurring in an OWL-QL query is considered to be a *must-bind variable*, a *may-bind variable*, or a *don't-bind variable*. Answers must provide bindings for all the must-bind variables, may provide bindings for any of the may-bind variables and should not provide bindings for any of the don't-bind variables. These designations are made by including a must-bind variables list, a may-bind variables list and a don't-bind variable list in an OWL-QL query. These lists contain URI references occurring in the query, while no URI reference can be an item of more than one of these lists. Consider a query with the query pattern "{(hasMother ?p ?m)}", meaning "?p has mother ?m", and the following cases:

- **?m is a don't-bind variable**: The complete set of query answers contains one answer for each known person that identifies a person but not the person's mother.
- **?m is a must-bind variable**: The complete set of query answers contains one answer for each known mother that identifies both a person and the person's mother.
- **?m is a may-bind variable:** The complete set of non-redundant query answers contains one answer for each known person that identifies a person, while the person's mother is identified only when the mother is known.

The original query corresponding to the query pattern of Figure 8, as posed by the querying client with an answer bundle size bound of 5 an the ?p variable, that represents the black car owners, considered as a must-bind variable is shown in Figure 9.

A query may have zero or more answers, each of which provides bindings of URIrefs or literals to some of the variables in the query pattern. Each binding in a query answer is a URIref or a literal that either explicitly occurs as a term in the answer or is a term in OWL. A variable that has a binding in a query answer is identified in that query answer. The answer "Matt owns a black car.", as sent by the answering server to the query of Figure 9 is shown in Figure 10.

2.5 SWQL

The Semantic Web Query Language (SWQL) [42, 28], (pronounced 'swequel') serves to query all kinds of data that occur within the Semantic Web, namely RDF [34] and XML. Furthermore,

[1] A query pattern is represented as a set of triples of the form (<property> <subject> <object>), where any item in the triple can be a variable. Variables are shown as names beginning with the character "?".

```
<owl-ql:query xmlns:owl-ql="http://www.w3.org/2003/10/owl-ql-syntax#"
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#">
<owl-ql:queryPattern>
<rdf:RDF>
<rdf:Description rdf:about="http://www.w3.org/2003/10/owl-ql- variables#p">
<owns rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#c"/>
</rdf:Description>
<Car rdf:ID="http://www.w3.org/2003/10/owl-ql-variables#c">
<has-color rdf:resource="#Black"/>
</Car>
</rdf:RDF>
</owl-ql:queryPattern>
<owl-ql:mustBindVars>
<var:p/>
</owl-ql:mustBindVars>
<owl-ql:answerKBPattern>
<owl-ql:kbRef rdf:resource="http://chrisadata/chrisa.owl"/>
</owl-ql:answerKBPattern>
<owl-ql:answerSizeBound>5</owl-ql:answerSizeBound>
</owl-ql:query>
```

Figure 9: The Query posed by the querying client

it keeps the concrete data types and syntax transparent to the user by only relying onto the RDFS/OWL [46, 15], ontology vocabulary.

SWQL could be of great value in the emerging Semantic Web, where the same kind of information is represented in different syntactic forms, being either incompatible (XML or RDF) schemas or XML versus RDF. SWQL keeps these syntactic differences transparent to the user, so that the same query will work on both RDF and XML data, even with different schemas.

The syntax of SWQL is based on XQuery, but the underlying semantics are significantly changed. The main difference is the new data model. The SWQL data model is based on a graph structure and is therefore very similar to the RDF data model.

Another feature changed in SWQL is the path language. A new path language, *SWQLPath*, has been defined. SWQLPath is very similar to XPath [12] regarding steps and filters, but uses different tests. It is also capable of navigating through the data model and selecting objects out of it. The example path expression shown in Figure 11 selects the titles of all the resources that are exhibited:

The two main constructs in SWQLPath are the so-called NodeTests and PropertyTests. *NodeTests* filter a set of nodes for being an instance of a given OWL class. *PropertyTests* are used for navigation via the edges of the graph. They check if any node in a set of current nodes has a property that is an instance of the given property type and if yes, they return the corresponding node in the range of the property. As in XPath, SWQLPath supports also the so-called Predicates. *Predicates* filter a set of nodes based on some condition.

Besides path expressions, most XQuery expressions are also available in SWQL, namely FLWOR expressions, conditional (if-then-else) expressions, existential and universal quantification, user-defined functions, type switches and casts.

The third feature that is changed in SWQL with respect to XQuery is the type system. SWQL uses a completely new type system that is based on the emerging OWL standard. This type system is used to formulate syntax-independent queries by only checking type relationships with an inference system.

In the following, we will focus on the SWQL type system and the SWQL data model.

```
<owl-ql:answerBundle xmlns:owl-ql=http://www.w3.org/2003/10/owl-ql-syntax#
xmlns:var="http://www.w3.org/2003/10/owl-ql-variables#">
 <owl-ql:queryPattern>
 <rdf:RDF>
<rdf:Description rdf:about="http://www.w3.org/2003/10/owl-ql-variables#p">
<owns rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#c"/>
</rdf:Description>
<Car rdf:ID="http://www.w3.org/2003/10/owl-ql-variables#c">
<has-color rdf:resource="#Black"/>
</Car>
</rdf:RDF>
</owl-ql:queryPattern>
<owl-ql:answer>
<owl-ql:binding-set>
<var:p rdf:resource="#Chrisa"/>
</owl-ql:binding-set>
<owl-ql:answerPatternInstance>
<rdf:RDF>
<rdf:Description rdf:about="#Chrisa">
<owns rdf:resource="http://www.w3.org/2003/10/owl-ql-variables#c"/>
</rdf:Description>
<Car rdf:ID="http://www.w3.org/2003/10/owl-ql-variables#c">
<has-color rdf:resource="#Black"/>
</Car>
</rdf:RDF>
</owl-ql:answerPatternInstance>
</owl-ql:answer>
<owl-ql:continuation>
<owl-ql:none/>
</owl-ql:continuation>
</owl-ql:answerBundle>
```

Figure 10: The Answer returned by the answering server for the OWL-QL query of Figure 9

2.5.1 Type System

The type system of SWQL is based on OWL. This means that all queries are posed in terms of an implicit or explicit given ontology. Every class or property defined in such an ontology has a unique qualified name. These names are used in the query to refer to the corresponding class or property.

There is one function defined for class and property names, the "subsumes" function that checks for subsumption of two classes or two properties. In the example of Figure 12, if class1 subsumes class2 the "subsumes" function returns true, otherwise false.

2.5.2 Data Model

OWL assumes a graph-based data model as opposed to the tree-based data model of XQuery. The formal definition of the data model for OWL instances as used by SWQL, according to Figure 13, is as follows: All the parts of a graph are called items. Items are either nodes or properties. A node is either an object or a literal. Every *Item* of a graph has an associated *type* from the OWL ontology. The "type" function returns this associated type as a qualified name as defined in XML Schema. Every *Property* connects two nodes. The "domain" of a

```
Resource()/exhibited/title
```

Figure 11: Example SWQL Path Expression

```
subsumes($class1 as xs:QName, $class2 as xs:QName) as xs:Boolean
```

Figure 12: Subsumption Example

Property is always an *Object*, the "range" can be any *Node*. All nodes in a graph may have Properties that can be returned by the "toProperties" function. For *Literals* this set is always non-empty. All Objects may have Properties that can be returned by the "fromProperties" function. Every Literal has a value, which is an atomic type as defined in the XQuery 1.0 and XPath 2.0 specifications. Finally, the function "all" returns a set of all the Nodes existing in the graph.

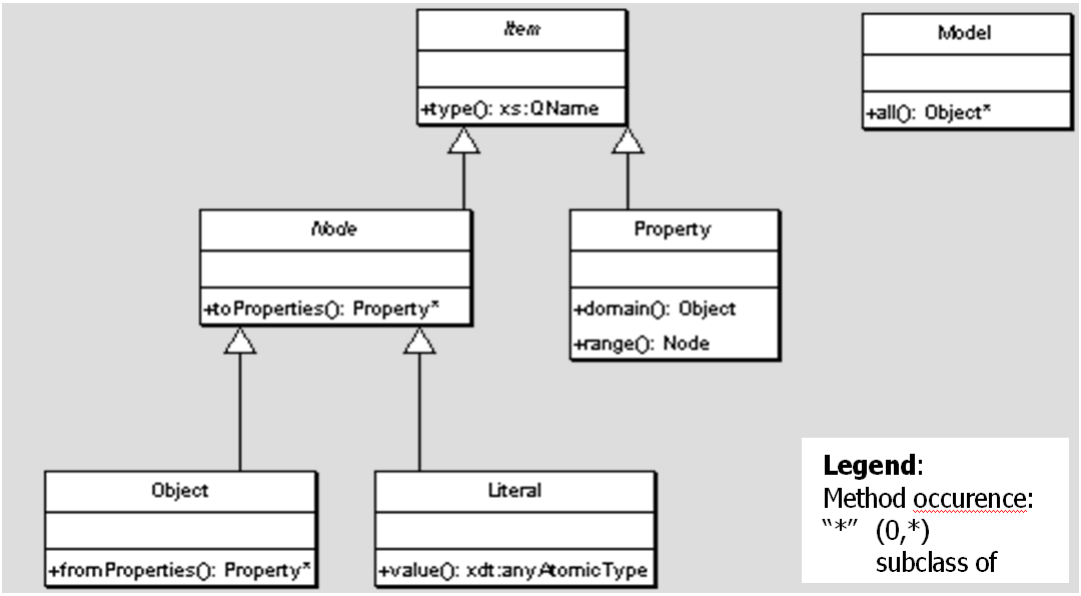

Figure 13: The SQWL Data Model

3 Query answering techniques

The large power and flexibility of the query languages we analyzed in the previous section give rise to several research issues in effectively solving the query answering problem. In this section we will first give an overview of the problem of structural query matching evaluation, then we will deepen the analysis of approximate query answering, which is a central issue in the Wisdom scenario. In particular, in this context, we will analyze available structural query rewriting techniques which were proposed and developed in peer data management systems involving ontology data but also in systems managing heterogeneous XML document bases; such techniques could be fruitfully adapted to the complex architecture of heterogeneous/distributed ontologies of Wisdom.

3.1 Exact versus approximate query answering

Differently from standard text retrieval queries, queries submitted to structure-aware search engines have to match the data (ontologies or simple XML documents) not only in their *contents*

(semantics) but also in their *structure* (the tree or, generally, graph describing the particular nesting of the elements). Such a match can either be *exact* or *approximate*.

As far as *exact matching* is concerned, the goal of evaluating tree pattern queries, such as the ones expressed in XQuery, sometimes called the *twig queries*, is to find all existing ways of embedding the pattern in the data. Since XML data collections can be very large, extensive research is currently being done in devising efficient evaluation techniques for tree pattern matching. Though XML data objects and queries are ordered labeled trees (*ordered tree matching*), in the majority of situations the user would prefer to consider query trees as unordered. In general, the process of *unordered tree matching* is difficult and time consuming. For example, the *edit distance* on unordered trees was proven *NP* hard in [63]. To improve efficiency, an approximate searching for nearest neighbors, called ATreeGrep, was proposed in [57]. However, the problem of unordered twig pattern matching in XML data collections has been studied only very recently. In [62], an efficient evaluation strategy for unordered tree matching is presented, exploiting the tree signature approach [61], which has originally been proposed for the ordered tree matching.

Exact matching strategies are important in order to answer queries expressed in query languages such as XQuery, however they alone are obviously insufficient to provide effective query answering to users in the complex architecture of heterogeneous/distributed ontologies of Wisdom. Indeed, the features of those languages allow for structural inquiries and introduce a significant complexity in the querying process. A user does not know the exact structure and concepts of all the available ontologies. Moreover, similar concepts coming from different sources typically use completely different structures and names. For all these reasons, it is necessary to go beyond exact matches and to consider techniques which are able to solve the complex problem of *approximate matching* (*approximate query answering)*, where the approximation has to involve both the structure and the contents of the ontologies.

Recently, several research efforts focus on the problem of *approximate query answering* in a distributed ontology scenario or, even more typically, in a conceptually similar heterogeneous XML document repository one. Much research has been done on the instance level, trying to reduce the approximate structural query evaluation problem to well-known unordered tree inclusion [54] or tree edit distance [36] problems directly on the data trees. However, the computational complexity of such approaches is nearly prohibitive for a realistic implementation. On the other hand, a large number of approaches prefer to address the problem of approximation and structural heterogeneity by first trying to solve the differences and discover correspondences between the ontologies / schemas on which data are based. *Schema matching* has been the focus of work since the 1970s in the AI, DB and knowledge representation communities and many algorithms and systems have been proposed, such as Similarity Flooding (SF) [35], providing particularly versatile graph matching algorithms. After having discovered the similarities between the schemas, it is possible to exploit the discovered correspondences in order to perform the operation of *query rewriting* (also known as *query reformulation*), which also in the Wisdom framework has a basilar importance and to which the following section is dedicated.

3.2 Query rewriting

In the *query rewriting* operation the submitted query is automatically "approximated" and rewritten in order to be compliant with all the available data. Query rewriting has been successfully exploited in several situations, in particular for improving the effectiveness of search engines querying distributed and heterogeneous data. Some of the techniques we describe in this section have not been originally conceived for working with ontology data but with heterogeneous XML document repositories, while others were conceived in a distributed peer data management systems setting involving ontology data. However, in any case, all such techniques are general and independent from the underlying matching discovery ones. Their only require-

ment is the availability of the correspondences between the schemas / ontologies. In Wisdom, the semantic mappings among the ontologies will be extracted by means of ad-hoc techniques and, thus, will constitute an optimal starting point for applying such existing query rewriting techniques. Indeed, with little changes, these techniques could be exploitable for rewriting the queries posed in the Wisdom project, originally expressed with reference to a local ontology, and could possibly be improved in order to exploit the full potential and expressiveness of the extracted semantic mappings.

3.2.1 The XML S^{3}MART approach

In [52, 45] a novel approach for query rewriting (and schema matching) has been proposed in order to solve the problem of effective and efficient search, by means of XQuery queries, among large numbers of "related" XML documents. The implementing system is named XML S^{3}MART (XML Semantic Structural Schema Matcher for Automatic query RewriTing) and its functionalities have been succesfully exploited in enhancing search effectiveness in heterogeneous XML document bases.

The approach relies on the information about the structure of the available XML documents. Due to the intrinsic nature of the semi-structured data, all such documents can be useful to answer the query only if, though being different, the schemas of the involved documents share meaningful similarities, both *structural* (similar structure of the underlying XML tree) and *semantic* (employed terms have similar meanings). For instance, the XML document shown in Figure 14 would clearly be useful to answer the FLWOR query of Figure 1. However, since its structure and element names are different from the ones assumed in the FLWOR expression, it would not be returned by a standard XQuery search engine. The *schema matching* process

```
<cdStore>
<name>Music World Shop</name>
<cd>
<vocalist>Elisa</vocalist>
<cdTitle>Then comes the sun</cdTitle>
</cd>
...
</cdStore>
```

Figure 14: XML Document that could not be processed using the FLWOR expression of Figure 1, although it is semantically relevant

proposed in [52, 45], extracts the similarities between the elements, which are then exploited in the proper query processing phase where the *rewriting* of the submitted queries is performed in order to make them compatible with the available document structures. The queries produced by the rewriting phase can thus be issued to a "standard" XML engine. For instance, the XQuery of Figure 1 can be automatically rewritten as shown in Figure 15, making it fully compatible with the previously shown document.

```
for $x in /cdStore
where $x/cd/vocalist = "Elisa"
return $x/name
```

Figure 15: The FLWOR expression of Figure 1 rewritten to handle the XML document structure of Figure 14

Matching and rewriting are performed on the actual structure of the XML data, where elements and attributes are identified by their full paths and have a key role in XQuery FLWOR

expression paths. By exploiting the best matches provided by the matching computation, a given query, written w.r.t. a source schema, can be straightforwardly rewritten on the target schemas. Each rewrite is assigned a score, in order to allow the ranking of the results retrieved by the query. The approach supports conjunctive queries with standard variable use, predicates and wildcards. After having substituted each path in the `WHERE` and `RETURN` clauses with the corresponding full paths and then discarded the variable introduced in the `FOR` clause, the query is rewritten for each of the target schemas in the following way:

1. all the full paths in the query are rewritten by using the best matches between the nodes in the given source schema and target schema (e.g. the path `/musicStore/storage/cd/singer` of source schema is automatically rewritten in the corresponding best match, `/cdStore/cd/vocalist` of target schema);

2. a variable is reconstructed and inserted in the `FOR` clause in order to link all the rewritten paths (its value will be the longest common prefix of the involved paths);

3. *a score* is assigned to the rewritten query. It is the average of the scores assigned to each path rewriting which is based on the similarity between the involved nodes, as specified in the match.

3.2.2 The Piazza PDMS approach

In [60, 37], the authors present a query rewriting (or reformulation, as they call it) technique in the context of the Peer Data Management System (PDMS) Piazza. In such a scenario every peer is associated with a schema that represents the peer's domain of interest, and semantic relationships between peers are provided locally between pairs (or small sets) of peers. The key step in query processing in a PDMS is reformulating a peer's query over other peers on the available semantic paths. The PDMS starts from the querying peer and reformulates the query over its immediate neighbors, then over their immediate neighbors, and so on. Whenever the reformulation reaches a peer that stores data, the appropriate query is posed on that peer, and additional answers may be found. Since peers typically do not contain complete information about a domain, any relevant peer may add new answers. Furthermore, different paths to the same peer may yield different answers.

Peers in a PDMS are linked through peer mappings. Peer mappings in Piazza are described as query expressions using a subset of XQuery. Figure 16 shows an example of a Piazza mapping. The mapping defines the relationship between the schema of S2 and the schema of S1; the two schemas differ in how they represent the advisor/advisee information. Schemas are represented using a format in which indentation indicates nesting and a * suffix indicates an arbitrary number of occurrences of the subelement. Two cases are contemplated for reformulation:

- *Single-step reformulation*: Suppose a query Q is posed over the peer P1. If P1 contains its own data, then the PDMS first retrieves the answers to Q based on P1's data. Then, Q is reformulated on P1's neighbors and appropriate queries are posed to them, and the process continues recursively.

- *Multi-step reformulation*: A PDMS answers queries by chaining individual reformulation steps. In this way, a PDMS can follow arbitrarily long semantic paths and retrieve answers from peers not directly connected to the query peer. Given a single-step reformulation algorithm, a template algorithm for query answering in a PDMS is implemented iteratively as follows. Suppose that the PDMS includes only pairwise mappings. At every point, a tree of goals G, each of which is a query on a particular peer, is maintained. Initially, G includes a single goal with the original query and peer, i.e., (Q,P). At each iteration, one of the leaf goals (Q'; P') $\in$ G is chosen. First, if P' contains data, the query Q' is posed

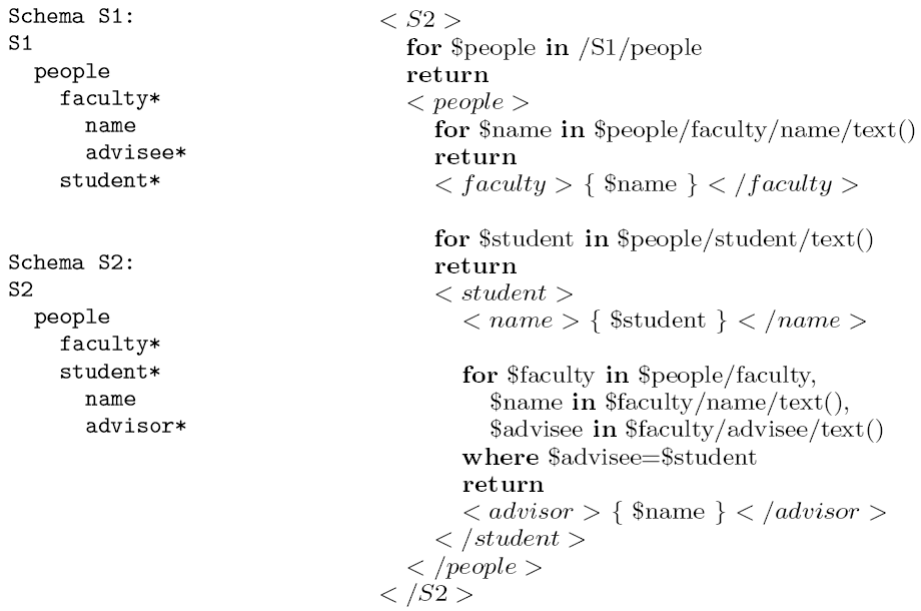

Figure 16: An example pair of peer schemas (left) and a Piazza mapping between them (right) used for rewriting queries

on the peer P' and the set of answers is added to the set of answers to Q. Second, Q' is reformulated on all the neighbors of P' and the newly created goals are added to G.

Finally, in [37] many optimizations for the reformulation problem are also proposed, since following all semantic paths naively may lead to several inefficiencies. The optimizations are related to:

- Pruning and minimization: algorithms for pruning redundant reformulation nodes and for minimizing reformulations are proposed.
- Search strategies: since reformulation can be viewed as a search through a space of reformulations, the effects of the search strategy on the reformulation time are studied in detail.
- Pre-computing semantic paths: the authors show that pre-computing certain paths in the network can offer many benefits, and examine the tradeoffs involved.

3.2.3 The Xyleme approach

Xyleme [19], which stores data from multiple sources in a warehouse, uses an elaborate approach to discovering and defining mappings, based on path-to-path mappings. The Xyleme query language is an extension of OQL with path expressions and the full text `contains` predicate. Figure 17 shows a sample query that retrieves the titles of van Gogh's paintings exposed at the Orsay museum.

A Xyleme view has a domain, a schema and a definition. Everything related to the view domain is called concrete and everything related to the view itself is called abstract. The domain of a view is a set of clusters, i.e., a sub- set of Xyleme repository. The structure of the documents within a cluster is described by DTDs. The view schema is an abstract DTD. It is a tree of concepts (rather than attributes or elements). A view is defined by a set of pairs $< p, p' >$, called mappings, where p is a path in the abstract DTD and p' a path in some concrete DTD. The intuition underlying Xyleme views is that of path-to-path mapping, i.e. a view specifies mappings between path in the abstract and concrete DTDs. Query Translation is performed in

```
select p/title
from doc in culture,
p in doc/painting,
where p/author contains "van Gogh"
and p/museum contains "Orsay"
```

Figure 17: An example query in Xyleme

two steps. First, the abstract concepts that are used in the query are selected. For instance, culture/painting, culture/painting/title, culture/painting/author and culture/painting/museum in the query of Figure 17. Then, among all the possible ways of combining their associated concrete paths, only those that (i) form a subtree of some concrete DTD and (ii) preserve the parent/child relationship of the abstract query are selected. The result is the union of the concrete queries built from the valid tree combinations. Abstract to Concrete Translation (A2C) algorithm transforms abstract pattern trees into concrete ones (see Figure 18 for an example). To compute the translation of a whole tree, it is decomposed in upward paths starting from each leaf and stopping when a node that has already been visited by a previous upward path is reached. Once the decomposition has been performed, A2C translates each upward path to a concrete branch, then it computes concrete pattern trees by combining branch solutions.

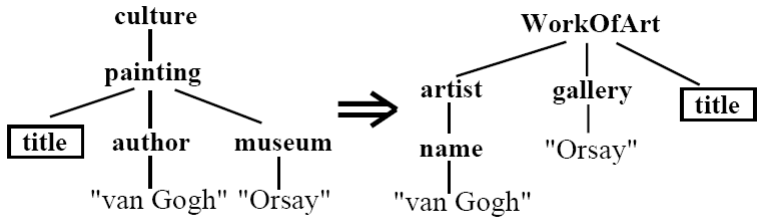

Figure 18: An example of abstract to concrete query tree reformulation in Xyleme

3.2.4 Other approaches

Many other query rewriting / reformulation approaches exist in the literature, especially in a standard relational scenario, which, however, is significantly different from the one of the Wisdom project. Therefore, in this paragraph we will only present very shortly two of the most significant of such approaches, from which some basic ideas could nonetheless be borrowed.

In [20] a system called MARS (Mixed And Redundant Storage) for publishing XML data from mixed (XML+relational) proprietary storage is presented. Client queries, as well as GAV and LAV views are expressed using the standard XQuery language, and the MARS sustem is able to find the minimal query reformulations, i.e. the system guarantess that there is no redundant data source scan. In particular, the navigation part of XQueries is reformulated. In order to achieve that, MARS uses a compilation of queries, views and constraints from XML into the relational framework. The compilation reduces the original reformulation problem to a problem of minimization of relational queries under relational integrity constraints.

The work [50] proposes PeerDB, a P2P-based system for distributed sharing of relational data. Similar to the Piazza system (see section 3.2.2), PeerDB does not require a global schema but it does not use schema mappings for mediating between peers. Instead, PeerDB employs an Information Retrieval based approach for query reformulation. A peer relation (and each of its columns) is associated with a set of keywords. Given a query over a peer schema, PeerDB reformulates the query into other peer schemas by matching the keywords associated with the two schemas.

4 Ontology visualization and exploration tools

One of the open research problems in the field of ontology management is to find a well-suited formalism and a proper user-interaction paradigm giving end-users an *intuitive* representation of knowledge and a *friendly* tool for collecting it.

Intuitive means that the visual language used to graphically represent information must be able to highlight (an 'interesting' part of) its semantics in such a way to make the drawn picture *significant* to end-users: the set of graphical objects that could appear in the picture define a notation that is more direct and easy-to-interpret than the formal definition of the described entities (usually expressed by means of an abstract syntax), still being capable of unambiguously expressing their meaning.

Friendly means that the interaction paradigm supplies a few navigation primitives that allow users to expand their view on a particular knowledge base, in both a conceptual and a graphical sense, through *simple* actions performed on the visual encoding of that knowledge. Typically, each class of user actions (e.g. mouse click, drag & drop, ...) triggers a different kind of query on the underlying query engine: independently of their form, answers are then given a visual rendering users can exploit for further retrieval of new (usually related) information.

Following this pattern, research in the field of *knowledge presentation* has recently led to the development of several visual languages, user interaction paradigms and corresponding tools as a headway to industrial practice; nevertheless, the goal still seems to be far from being reached. In this section, our investigation has the purpose of reporting results that have originated from this work by an evaluation of the currently available tools. As each tool is examined under two different aspects, namely the representation formalism and the user-interaction pattern, for both we define a set of evaluation criteria according to which a classification scheme is set up.

4.1 Evaluation Criteria

For the evaluation of knowledge presentation tools, it is crucial to recognize that one of the most challenging research issues in this area consists in finding the optimal trade-off between the domain-independence of the notation and the expressivity of the visual language.

As the visual language includes more and more expressive constructs, i.e. as the meaning of those constructs gets more and more precise (and significant to end-users bound to a particular context), it suffers from an increasingly higher loss of generality; this is due to the fact that a very expressive graphical notation defines *ad hoc* representations for a restricted set of entities, thus becoming tailored to those application domains in which such entities exist. Figure 19 illustrates this situation.

The same consideration applies to navigation primitives: general-purpose presentation tools would supply end-users with a set of operations such as "view classes", "view instances", "hide subclasses", "show superclasses", and so on, thus supporting the exploration of every kind of knowledge base; instead, dedicated presentation tools tightly bound to any application domain would associate user actions with more specific queries, like "retrieve written books", "find movies", "get neighbor nations", and so on.

Intuitively, domain-independent formalisms are more reusable, but represent entities in a way that is 'closer' to the abstract syntax used to formally define them, thus being more difficult to understand for end-users; on the contrary, domain-specific formalisms have a lower degree of reusability, but define graphically richer constructs that allow for a representation that is 'closer' to how the corresponding entities appear in the described domain (that is, graphical properties of graphical objects are provided for each conceptual property describing the real-world entities those objects represent).

As an attempt to retain domain-independence while providing an expressive visual language for representing knowledge, a shift is currently being registered towards the development of flexible languages that allow for different notations and user interaction patterns to be selected

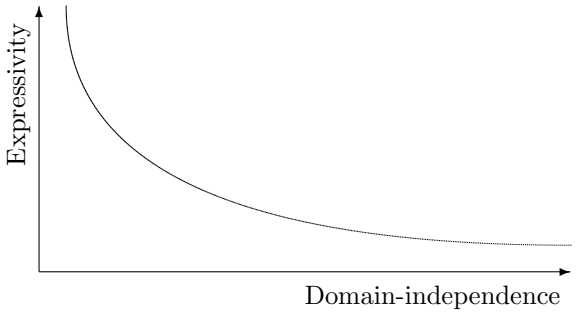

Figure 19: Trade-off between domain-independence and expressivity of the visual language

according to the end-user's needs, profile and context. A very effective form of flexibility is realized by allowing some parameters to be provided for the instantiation of a generic visualization formalism (for example, a visualization engine that accepts style sheets as formatting directives). In agreement with the terminology used in [25], we say that knowledge presentation tools providing such a framework are *configurable*. The concept of configurability applies both to representation languages and user interaction patterns: in short, configurable navigation primitives would allow users to specify which (kind of) query is associated to which (kind of) action.

As most of the ontology definition languages treat concepts and roles as, respectively, the equivalent of First Order Logic unary and binary predicates, an ontology can be both graphically and conceptually understood as a graph where nodes stand for concepts or individuals, while edges stand for property instances relating pairs of concepts, concepts to individuals, or pairs of individuals. Consequently, most ontology representation systems offer a graph-based visualization and navigation front-end. However, even when configurable systems would allow for style sheets to be provided in order to associate custom drawing directives for nodes and edges, abstractions that populate different application domains (or their most proper representation) could not always fit in a graph structure: thus, powerful tools for knowledge presentation should encompass different metaphors for representing such knowledge. We refer to tools providing this feature as *multi-metaphor* systems. Notice that not being a multi-metaphor system does not necessary implies that navigational and representational features are poor: for instance, very effective visualization techniques exist to make graphs more readable and easy-to-explore, such as clustering [13], zooming, panning, filtering, force-directed layout of nodes [49], the Graphical Fisheye views [33], the Rubbersheet views [53], and many others. As we will see in Section 4.2, configurable graph-based tools can do very well in presenting ontological information.

An interesting feature of multi-metaphor knowledge presentation systems is *orthogonality*. A fully orthogonal, multi-metaphor system allows each metaphor (navigation primitives and graphical constructs) to be selected for rendering and exploring any knowledge base (or any subset of the same knowledge base). Some non-orthogonal tools, for example, define completely different formalisms and navigation environments for exploring the schema and the instances of a given knowledge base. Depending on the peculiarities of the application domain and the profile of end-users, this could be either an advantage or a drawback: as reported in [25], there are cases where different parts of the same knowledge base are best represented by means of different visualization metaphors. In such situations, a multi-metaphor, but still configurable, system would not suffer from the lack of orthogonality.

In [47], three main user categories are identified as the *target* of knowledge presentation tools, namely end-users, Guru users (or knowledge engineers), and developers. End-users are meant to be people with experience in the domain described by the ontology. The same applies to Guru users and knowledge engineers, who are additionally supposed to be domain modelers because of their greater experience on the application domain. Finally, developers are those

who cope with implementation details of the knowledge base. Knowledge presentation tools are often targeted to a subset of those categories: ideally, most flexible tools should encompass all of these user profiles.

Presentation tools that are mainly addressed at knowledge engineers and ontology developers must offer additional *editing* functionalities besides those allowing end-users to navigate a knowledge base. Related features could be consistency-checking, classification, discovery of hidden relationships, and so on.

Apart from this, visualization tools could also allow users to manually arrange depicted items in order to meet some particular visualization criteria or, simply, their taste. Such a feature can be thought of as a limited, *non-semantic* editing functionality that aims at improving the user interface's usability. We say that such tools support *graphical editing*.

As to the navigation features provided by knowledge presentation tools, we make a further distinction between *uniform* and *non-uniform* systems: *uniform* systems adopt the same environment for displaying both those pieces of information end-users act on when formulating queries, and query results; conversely, *non-uniform* tools make use of non-integrated environments for formulating queries and showing query answers, usually relying on different formalisms for representing these two kinds of information.

Also, when considering navigational features, we distinguish between *semantic browsing* and *general query formulation*, the former being a special case of the latter. Semantic browsing consists in exploring a knowledge base by navigating from object to object following property links or, at most, by formulating queries that are always centered on one (represented) entity. Instead, general query formulation is meant to be a more powerful (and complex) way of either 'manipulating' a set of visual objects or typing in textual information (but both kind of interactions could be used at the same time) to compose low-level queries without requiring a profound knowledge of implementation details; for example, keyword-based searches fall into this category.

4.2 Tools

In this section we are going to draw an overview of the existing tools for exploring and visualizing ontological knowledge. Features of such tools are briefly discussed in the corresponding subsections. In this survey, we do not discuss those interfaces that allow to access and query knowledge bases without providing any kind of visual abstraction to end-users, since they are not significant to the WISDOM Project. Examples of such systems are online demos for formulating low-level queries on a repository of ontological knowledge bases and for displaying answers, usually presented in a very raw form (see, for instance, Sesame [16] for RQL, Inkling [48] for SquishQL, and Redland [11]).

4.2.1 Thin clients: Ontolingua and Ontosaurus Web GUIs

Web clients for the Ontolingua [30] and Ontosaurus [59] repositories are two examples of rudimental ontology navigation systems providing pure HTML interfaces for exploring a set of concepts organized in specialization hierarchies (also named *taxonomies*), together with their formal definitions. Due to the limitations imposed by both the adoption of a Web interface and by portability requirements forbidding the use of extensions to the standard HTML language, the user interface is indeed not very attractive, offers only poor interactivity and does not encompass feedback mechanisms that are typical for effective collaborative environments. A more detailed discussion of the drawbacks of this kind of tools can be found in [21].

Besides its limited support to knowledge visualization, the Ontosaurus Web client allows for both *semantic browsing* and *editing*. A screenshot of the Ontosaurus Web interface is shown in Figure 20. As can be noticed, the interface is split into several frames: the upper-left and the

right frame are almost equivalent and can be used for navigating at the same time two separate parts of the same taxonomies, or even two unrelated taxonomies.

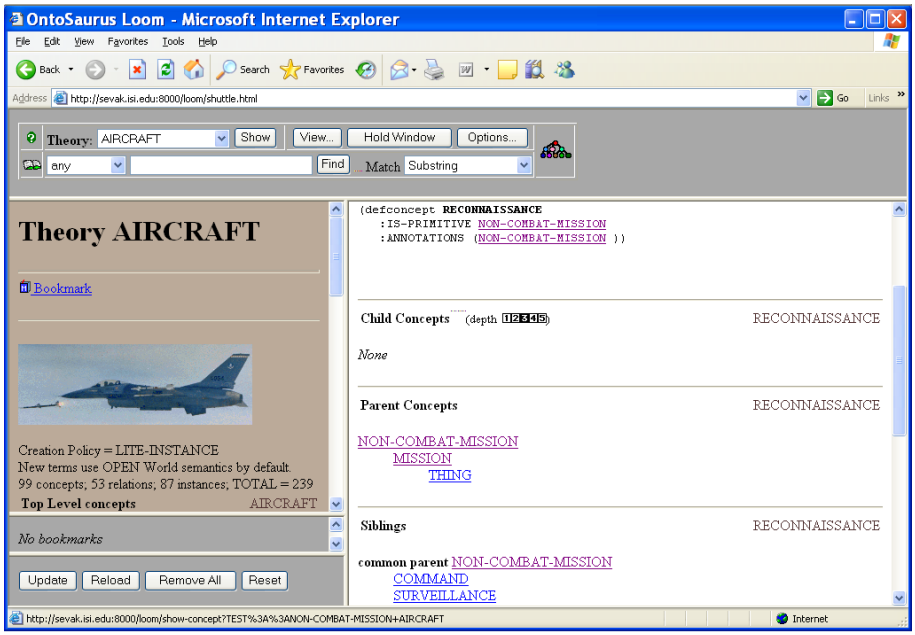

Figure 20: The Ontosaurus Web client interface

Navigation mainly consists in moving up and down in the specialization hierarchy, and a "Find Matching Instances" operation is provided to show direct and indirect instances of the currently selected concept. When instances are shown, a new page is loaded on the left frame containing a list of instance names. When instance names are clicked, a new form appears in the right frame reporting a short description, role fillers and classifying concepts for the selected instance. Thus, abstractions perceived by end-users are those of *concept*, *instance*, *subclass/superclass* and *role*, while the main navigation primitives are *show instances with properties* and *show types*.

The Ontolingua Web client has similar features, and will not be further discussed; Figure 21 illustrates a screenshot of the taxonomy browsing facility provided by Ontolingua's user interface.

The Ontosaurus and Ontolingua user interfaces are *not configurable*, meaning that no parameters can be specified in order to to highlight the semantics of some domain entities by changing the appearance of the corresponding displayed item. However, their general-purpose design allows any kind of ontological knowledge base to be explored, thus being highly domain-independent and (consequently) highly reusable. With reference to the user classification scheme introduced in Section 4.1, it is easy to recognize that such tools are mainly addressed at *domain modelers* and *ontology developers*, and, in general, to people who are quite familiar with a formal approach to knowledge engineering.

The Ontosaurus and Ontolingua Web clients could be considered as primitive *multi-metaphor*, *uniform* systems, in that two different visualization schemes and navigation primitives are integrated for the *semantic browsing* of concepts as well as instances. However, such systems are strictly *non-orthogonal* and *non-proactive*, because each presentation scheme can be applied only to one kind of entities. Finally, *no graphical editing* facilities are provided to manually adjust the rendering of the displayed information.

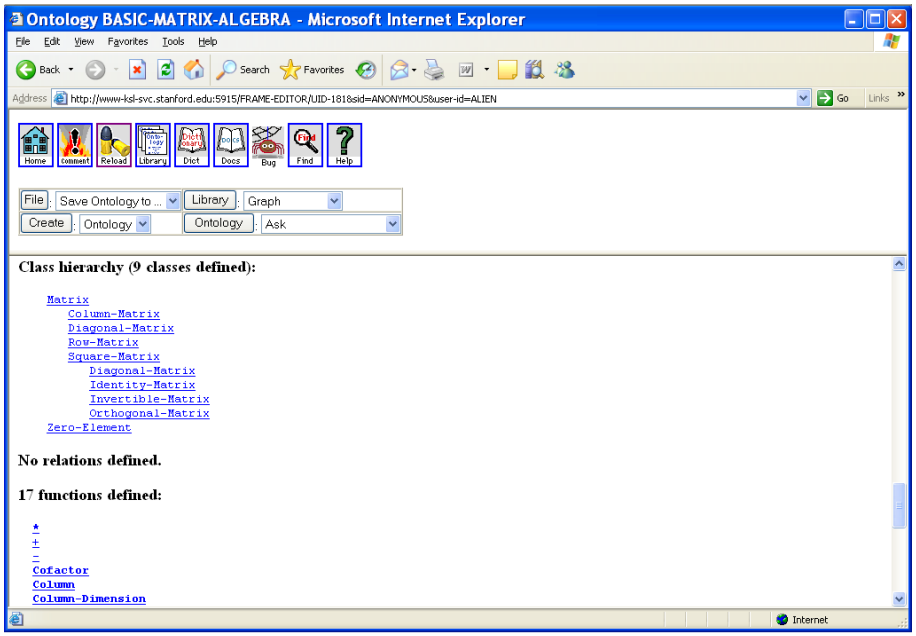

Figure 21: Part of the Ontolingua Web client interface

4.2.2 Thick client ontology editors: OilEd and Protégé

Ontology editors are a class of tools that need to implement intuitive knowledge base exploration techniques in order to let users examine both concept definitions and relationships holding between them, as well as edit new information. Also, effective facilities for visualizing instances and for browsing links between them are extremely important to this end. From now on, we will implicitly refer to *thick clients* only, whose GUIs are more complex and resource-consuming than those described in Subsection 4.2.1, but with a higher degree of interactivity and usability. Here, we present the user interface provided by two such tools, namely OilEd [10] and Protégé [4].

The former is a free ontology editor supporting DAML+OIL, developed by the University of Manchester, whose latest version also supports OWL; OilEd is able to reason on the $\mathcal{SHIQ}$ and the $\mathcal{SHF}$ Description Logics, by exploiting the services offered by the FaCT reasoning system [39]. Figure 22 illustrates part of the user interface provided by OilEd.

The latter is a free open-source Java tool based on the Eclipse [8] framework, providing an extensible architecture for the creation of customized knowledge-based applications. Indeed, Protégé is one of the most widespread tools currently used in the field of ontology management: to this end, it provides several plug-ins, including the Protégé OWL Plugin [5] for manipulating OWL, RDFS and RDF documents. Since the GUIs of these two products are very similar, both in the way they present ontological information and in the kind of offered *editing* functionalities, int this subsection we are going to discuss the user interface of the Protégé OWL Plugin only. A screenshot of the Protégé OWL Plugin GUI is illustrated in Figure 23.

As can be noticed, classes are shown on the left frame and are hierarchically arranged according to their specialization relationships. Actually, this is how most editing tools visually represent the semantics of *rdfs:subclassOf* property instances between different classes. Ontology editing tools usually give a visual rendering only to the semantics of those properties and classes that are part of the OWL, RDFS and RDF specifications, thus retaining a high degree of domain-independence. Since the *target* of such tools are knowledge engineers and developers, *configurability* is not the main concern, while the applicability range of the visualization metaphor and the user interaction pattern is a mandatory requirement.

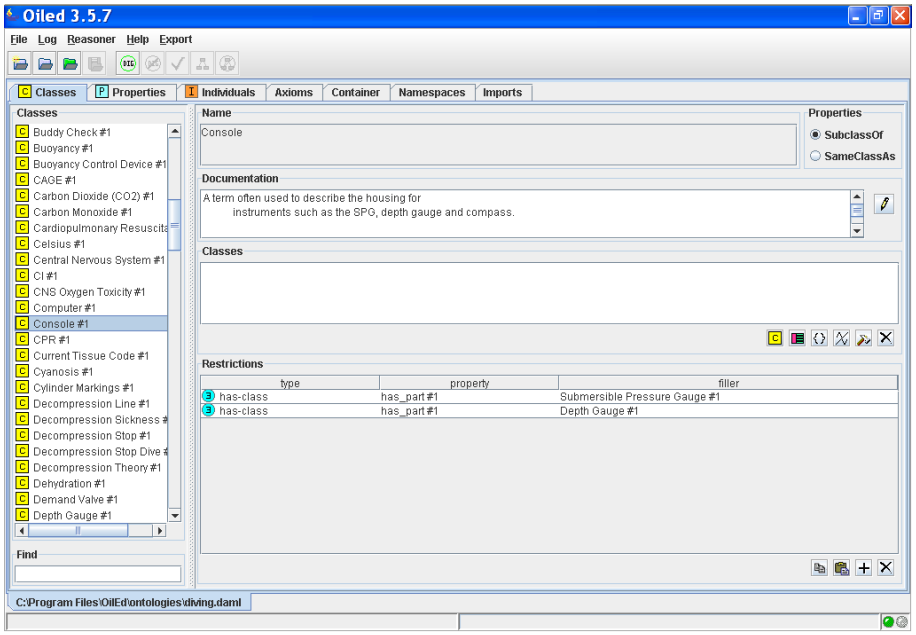

Figure 22: Part of the OilEd user interface

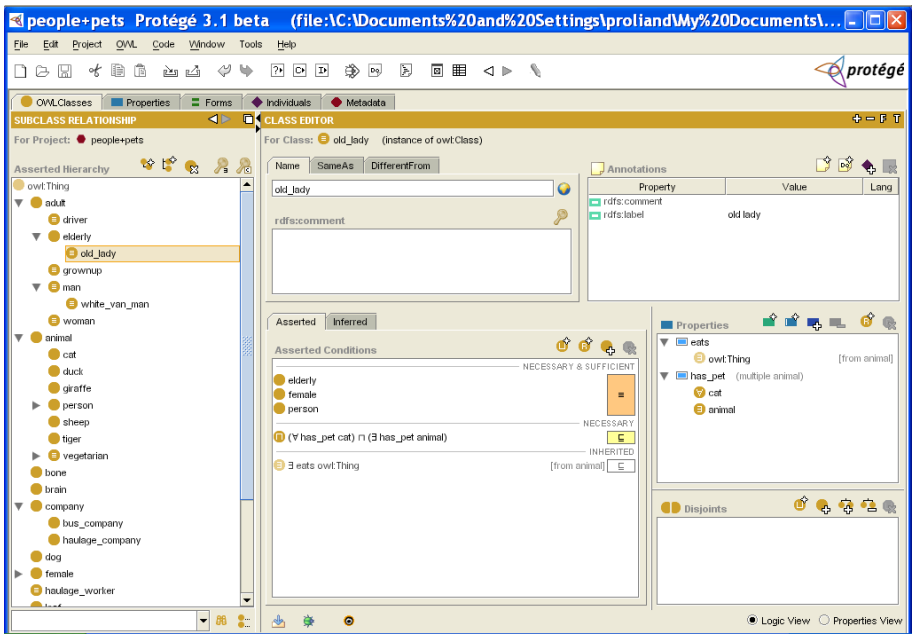

Figure 23: Part of the Protégé OWL Plugin user interface

In general, ontology editing tools provide two different visual languages for representing and editing the schema and instance subsets of an ontological knowledge base. As most knowledge bases draw a clear separation between such worlds, the presentation metaphor turns out to be usable in a wide range of situations. However, tools that are based on such an assumption often fail to manage the visualization and creation of knowledge bases where classes are treated as instances and viceversa, as allowed by the OWL Full specification.

Thus, the Protégé OWL Plugin has a *multi-metaphor*, *non-orthogonal* knowledge presentation feature. It also offers a limited form of *graphical editing* for customizing the way instances

are visualized in a form-like fashion: the kind of allowed customizations are mainly related to field repositioning and resizing.

Straightforward navigational features supported, including the well-known "expand subclasses" and "show instances" primitives, as well as a hyperlink-style browsing of properties relating pairs of individuals. Apart from this, a dedicated plug-in exists which allows to pose simple queries by typing textual information into some fields of a dedicated form.

4.2.3 Protégé Plugins

As already stated in Subsection 4.2.2, Protégé is an Eclipse-based framework supporting extensions for customized knowledge-based applications. This framework has been exploited for integrating tools explicitly oriented to knowledge visualization with other tools more specifically designed for ontology editing.

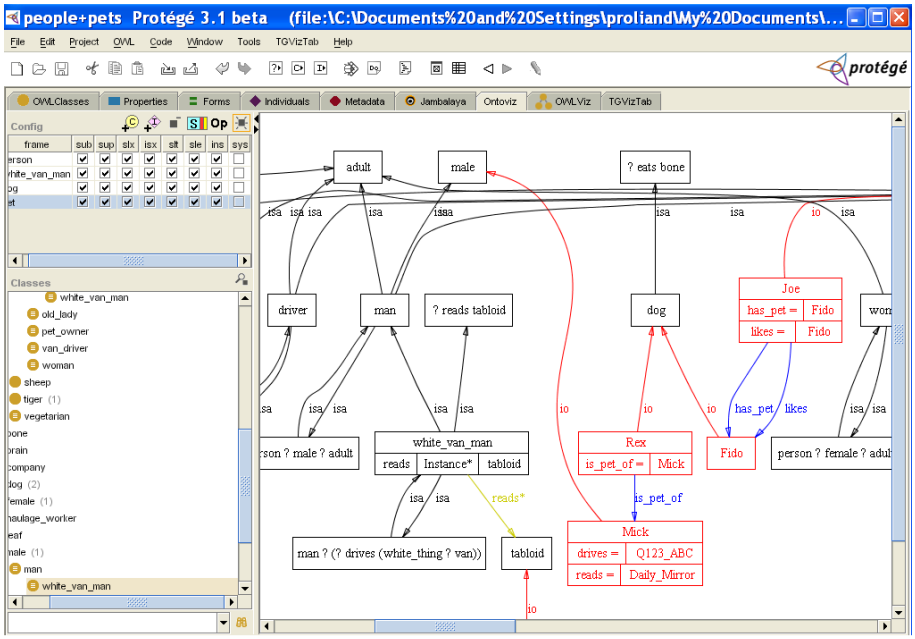

Figure 24: User interface of the OntoViz Plugin for Protégé

The OntoViz Plugin [2, 58] for Protégé makes use of the AT&T's GraphViz [1] tool to show an ontology as a graph-like structure, where classes and instances are rendered as nodes while property instances are drawn as edges. OntoViz is a *single-metaphor*, *configurable* tool, in that it only allows knowledge bases to be rendered as a single graph, but a lot of customization options are provided to change the visualization state of classes, properties, instances, property instances, as well as a way for specifying colors to use when displaying such entities. OntoViz does not offer *graphical editing* features, as the user is not able, for example, to perform drag & drop operations on graph nodes and, in general, to rearrange the graph in order to meet custom layout criteria. Figure 24 shows an example of how the OntoViz Plugin can render the same ontology shown in Figure 23.

The OWLViz Plugin [3] for Protégé , for which a screenshot is illustrated in Figure 25, is similar to the above mentioned OntoViz Plugin, in that it shows an ontology as an graph structure. Still, it has a very limited support for customization, by just allowing classes to be shown or hidden, and specialization relationships only are drawn. Thus, the OWLViz Plugin can be thought of as a *single-metaphor*, *non-configurable* tool mainly targeted at domain experts and knowledge engineers, providing *no graphical editing nor navigational* facilities.

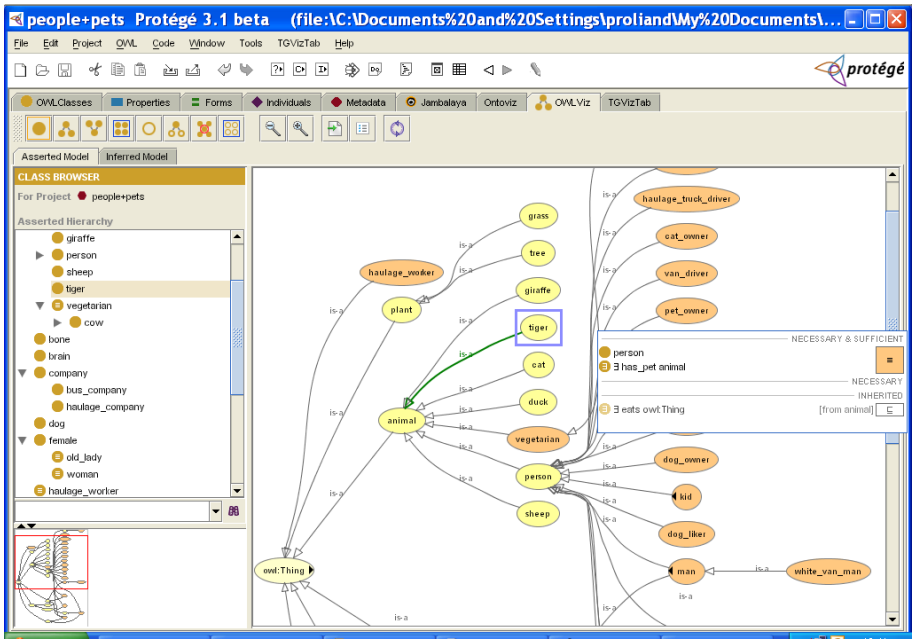

Figure 25: User interface of the OWLViz Plugin for Protégé

The TGVizTab Plugin [6] for Protégé differs from other graph-structured knowledge presentation systems because it is based on a smart visualization technique called *force-directed layout* [49]; the plug-in makes use of the TouchGraph [7] library to implement the force-directed layout algorithm. As usual, classes and instances are shown as graph nodes, while properties relating pairs of classes and instances are rendered as edges. The distinguishing feature of the force-directed layout approach is that the graph layout is automatically rearranged in such a way to make it as readable as possible, by minimizing the crossing of edges and the overlap of nodes.

The TGVizTab Plugin has *graphical editing* facilities, since it allows users to manually change the position of graph nodes: as drag & drop are being performed, the layout gets dynamically updated. As shown in Figure 26, however, the drawing of graphs containing many edges and nodes could result quite unwieldy even for users with deep knowledge of the rendered ontology. Fortunately, like the OntoViz Plugin, the TGVizTab Plugin offers a lot of customization options for changing the color used to draw different property and class instances and their visualization state, thus being a *single-metaphor*, *configurable* knowledge presentation systems.

As a final comparison, while the OntoViz and TGVizTab Plugins implement some simple navigation primitives mainly consisting of graph nodes expansion, OWLViz representations are completely static. This means that there is no way to obtain a representation for a different subset of the underlying ontology by acting on the currently available representation.

Jambalaya [24] is the last Protégé Plugin we discuss in this subsection. Designers of Jambalaya were motivated by a set of requirements collected by means of a broadly delivered questionnaire as described in [25], where the crucial conclusion is made that knowledge base management tools should be characterized by both *domain-independence* and *configurability* (also see Section 4.1 for a discussion of this issue). Jambalaya is an interactive, highly *configurable* environment for exploring ontological knowledge bases.

The main novelty in the approach Jambalaya represents ontological information consists in organizing graph nodes into containment hierarchies, where the semantics of such graphical containment can be specified by selecting the set of properties that determine when the visual representation of a resource should be arranged into the visual representation of another re-

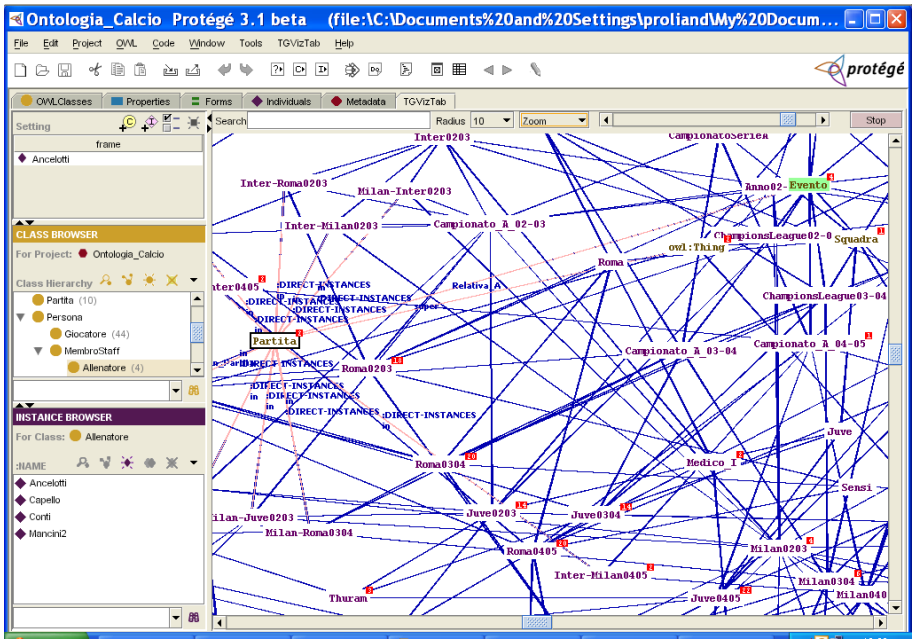

Figure 26: User interface of the TGVizTab Plugin for Protégé

source. By default, such properties are *rdfs:subclassOf* and *rdf:type*, meaning that *subjects* of *statements* whose *predicates* are *rdfs:subclassOf* or *rdf:type* must be rendered inside the visual items produced for *objects* of those statements (for an explanation of the terminology, refer to [41]).

As multiple containment hierarchies are generated, properties that relate objects from different hierarchies are traced as edges between the corresponding boxes. Thus, Jambalaya visually renders an ontology as a graph in which the semantics of some relationships, instead of being represented as edges, are encoded through graphical containment (see Figure 27).

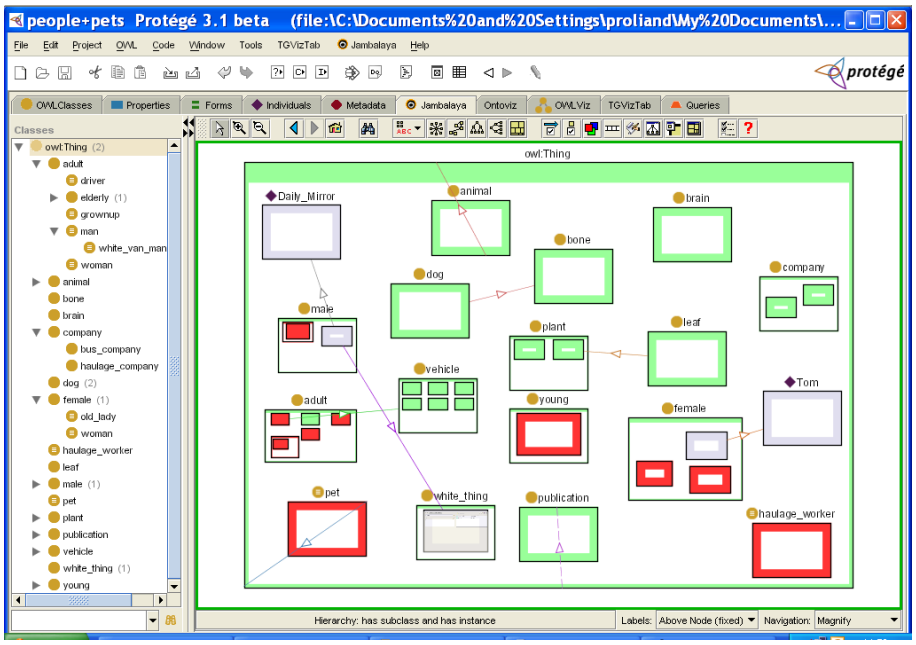

Figure 27: User interface of the Jambalaya Plugin for Protégé

Jambalaya also allows to display a form for editing the property values of any shown instance inside its representing node, together with zooming facilities for focusing one a subset of the ontology. Also, different colors can be associated to different classes of objects and different properties. Apart from the *graphical editing* facilities allowing users to manually rearrange the layout of the presented graph, Jambalaya only realizes a limited set of interaction primitives, which mainly consist in expanding the containment hierarchies and navigating property links.

4.2.4 WebOnto

WebOnto [22] is a Java applet coupled with a customized Web server which allows users to browse and edit knowledge models over the Web (WebOnto is now available as a public service). WebOnto makes use of the canonical visualization of ontologies as graphs (see Figure 28), and presents little innovation as compared to other existing tools. It is *not configurable*, not very interactive in the way users can refine their view on a given ontology (although some *graphical editing* is allowed, for example manual repositioning of nodes), and seems not to be a full-fledged knowledge exploration tool (indeed, WebOnto was designed to complement the ontology discussion tool Tadzebao [21] as an effort to enable collaborative browsing, creation and editing of ontologie).

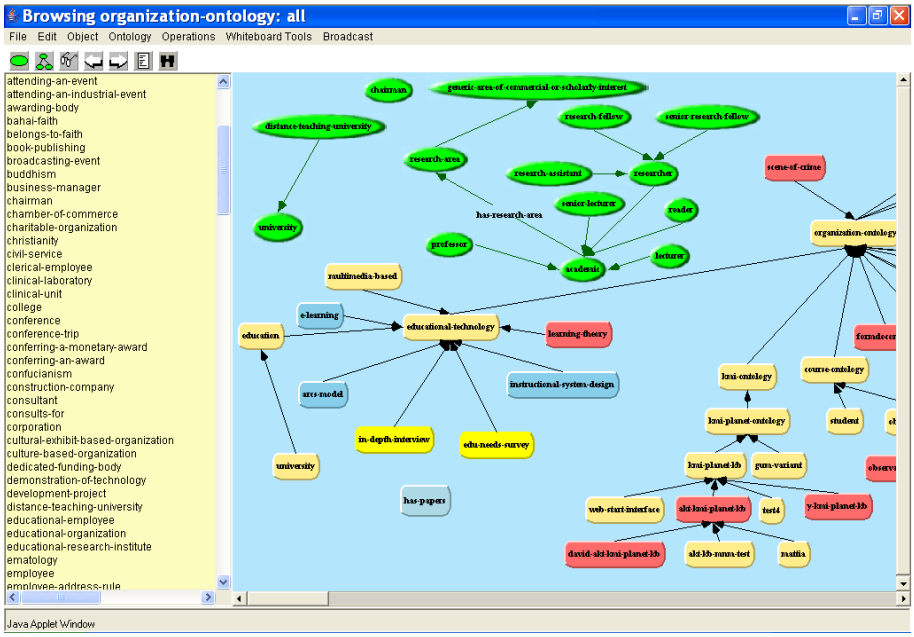

Figure 28: User interface of the WebOnto application

No *semantic browsing* facilities are provided, and ontology exploration is only performed by first selecting a concept from a (non-hierarchical) list, and by consequently drawing its specialization tree. Concept instances are shown in separate, form-like views where property values can be read and modified, thus making WebOnto a *non-uniform* knowledge presentation system.

4.2.5 Ontorama

Ontorama is another graph-based ontology presentation tool implementing the Graphical Fish-eye visualization technique [33]; this technique lets users focus on a restricted subset of the displayed ontology. As Figure 29 shows, the left frame contains a spherical distorted view that magnifies central elements while shrinking outermost ones (the magnifying lens metaphor is also

adequate). The right frame window allows to select concepts from a specialization hierarchy (taxonomy): when a concept is selected, the left frame view is automatically synchronized to put the corresponding element into the focus of the lens. The same effect can be obtained by double clicking on a node into the visualization sphere.

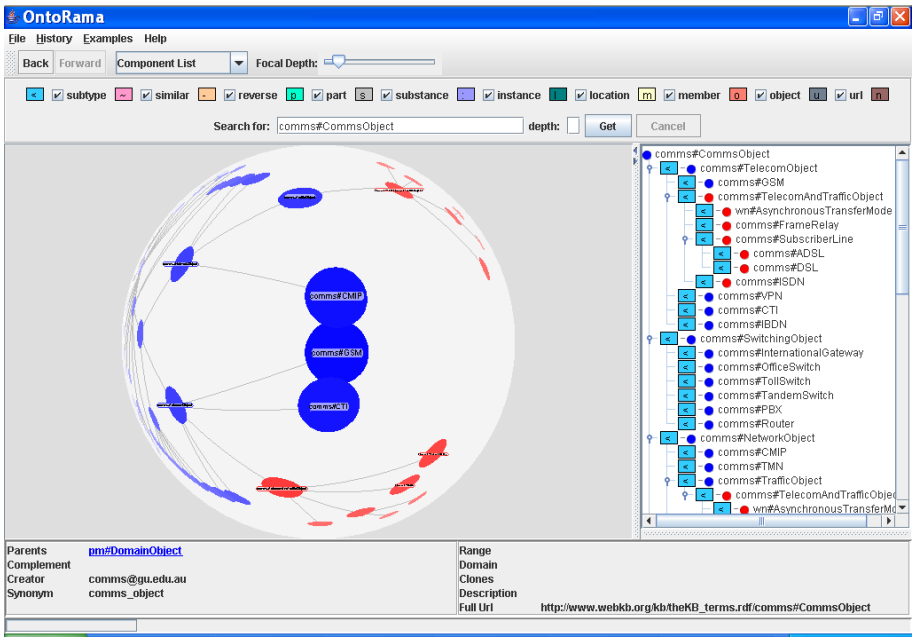

Figure 29: User interface of the Ontorama tool

The graphical arrangement of nodes in the view is performed automatically in such a way to make the graph as readable as possible, but no user intervention is allowed for manually modifying the generated layout. As for the TGVizTab Plugin for Protégé (see Section 4.2.3), since expressive filtering techniques are not provided, the obtained representation could become quite confusing as the size of the ontology gets larger. Finally, no navigation primitives are provided to move across different parts of the whole knowledge apart from transferring the focus of the lens from one concept to another.

Summing up, Ontorama is a domain-independent, *single-metaphor*, *non-configurable* software for the visualization of ontologies. Although Ontorama greatly benefits from the adoption of the Graphical Fisheye technique for focusing the view of a graph on a small set of nodes, it suffers from the lack of most of the other feature an exhaustive tool should expose. In particular, it does not support neither *semantic browsing* nor *graphical editing*.

4.2.6 KAON front-end

The Karlsruhe Ontology (KAON) tool suite has been developed in the context of the KAON Semantic Web infrastructure [43, 38]. KAON is an open-source ontology management infrastructure targeted for business applications.

It includes a comprehensive tool suite allowing easy ontology creation and management and provides a framework for building ontology-based applications. An important focus of KAON is scalable and efficient reasoning with ontologies. In this context, we are only interested in discussing the features of the Java-based front-end that is available at http://kaon.semanticweb.org/demos, for which a screenshot is illustrated in Figure 30.

Again, the Java-based front-end shows an ontology as a graph structure, and makes use of the force-directed layout algorithm to dynamically reposition graph nodes as node expansions are performed (a raw form of *semantic browsing*) in order to maximize the overall readability.

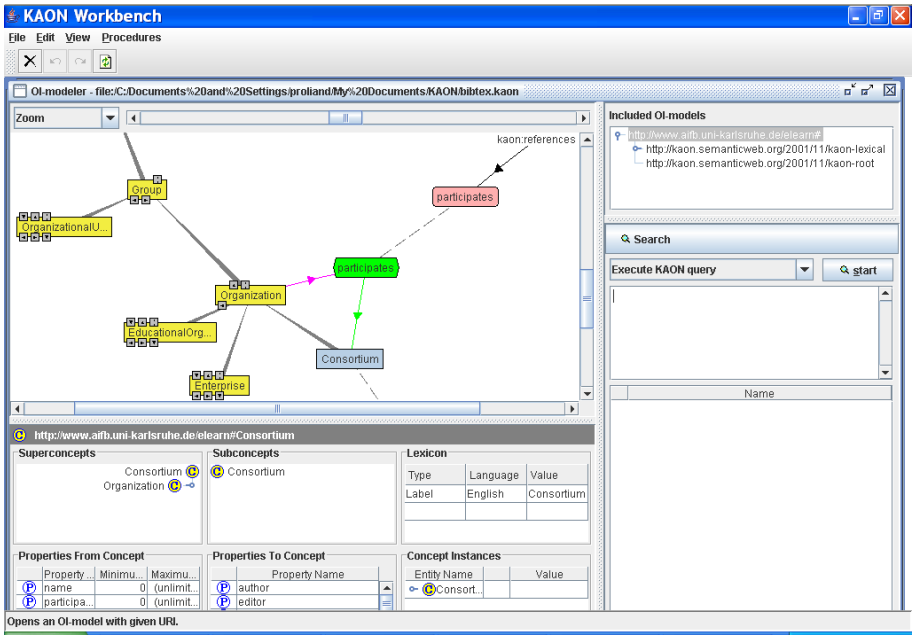

Figure 30: Java-based KAON front-end available online

Ontology *editing* is also supported, as new concepts, instances and properties can be easily created and added to the current view.

Apart from the ability of expanding and collapsing (as well as hiding and showing) graph nodes, the *query formulation* support is limited to keyword-based searches of concepts, properties and instances. *Configuration* options for the user interface do not encompass any way of customizing the appearance of nodes according to the semantics of denoted resources, e.g. by highlighting certain concepts or instances, associating different colors or drawing styles for edges that represent particular properties, and so on.

4.2.7 Spectacle

Spectacle [32] is a commercial tool allowing for a visualization of ontological information based on the Cluster Map metaphor. As for most of the previously described approaches, ontologies are still rendered as graphs: however, graph nodes are added here not only for single concepts, but also for pairwise intersections of concepts. Concepts and concept intersections are represented as sets (clusters) of elements with a finest granularity (instances).

Cluster Maps offer a significant view over a knowledge base by presenting an intuitive, detailed and quantified report of how instances are spread out through a set of classes chosen from a given taxonomy. A variant of the well-known spring-embedder algorithm for graph drawing [23] is used to make the depicted image as clear as possible. Cluster Map views are partially *configurable* as to the rendering style, but the only highlighted semantics are those of instantiation and specialization relationships, as well as concept intersection. Figure 31 shows a sample Cluster Map related to Job Vacancies (the source is [32]).

A limited form of *query formulation* is allowed, which consists in selecting a different set of classes from the given taxonomy according to which clusters are generated. A Cluster Map view over a given knowledge base could be refined by choosing subclasses of the actually selected ones to filter out uninteresting instances and clusters. *Semantic browsing* is possible in the sense that instances are shown as a textual list of individual descriptions when the user clicks on a cluster (thus making Spectacle a *non-uniform* system): however, no advanced visual facilities are provided for visually describing the semantics of the extensional part of an ontology.

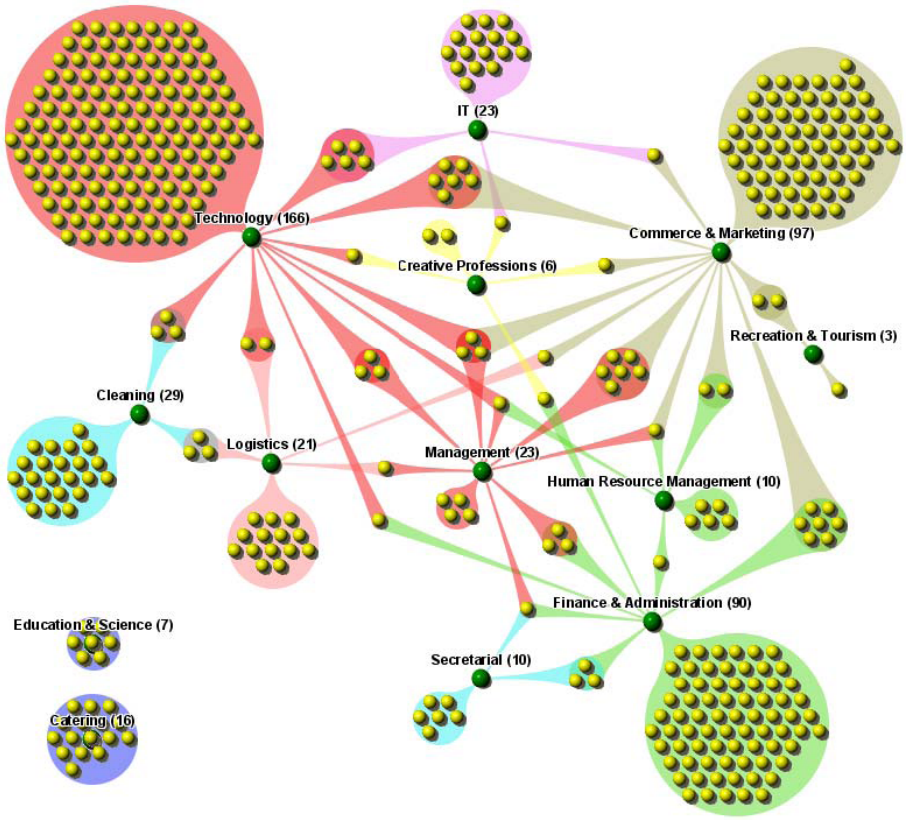

Figure 31: A sample Cluster Map

With reference to the classification of users drawn in Section 4.1, the Spectacle system is mainly targeted at end-users. Spectacle does not provide any *semantic editing* nor *graphical editing* features.

4.2.8 IsaViz and Graph StyleSheets

IsaViz is a W3C's visual environment for browsing and authoring RDF models represented as graphs. The user interface allows for smooth zooming and navigation in the graph creation, and for editing of graphs by drawing ellipses, boxes and arcs. The novelty of IsaViz is the introduction (from version 2.0 on) of Graph StyleSheets (GSSs) as a mechanism for associating style to semantics (the underlying idea is to give GSSs the same role CSSs have in the formatting of HTML documents): this gives IsaViz a very high degree of *configurability*.

As nodes denoting concepts and instances with certain properties can be rendered by means of icons that describe entities living in a particular application domain, images that are familiar to a restricted class of end-users, even bound to a specific application domain, could be used as an intuitive representation for entities belonging to that domain. Also, the graph metaphor for representing knowledge bases is not the only way information could be visualized: a tabular form for rendering concepts and instances is also available, as can be seen from Figure 32.

Since edges and nodes are processed by a generic formatting engine, which assigns styles according to the visualization directives contained in GSSs, this framework can instantiate domain-specific representations without suffering from a significant loss of reusability. Moreover, since GSSs can be understood as a primitive, declarative way of specifying visualization metaphors, *orthogonality* is an interesting characteristics of IsaViz.

The main drawback of this tool is the lack of any semantic browsing and non-trivial query

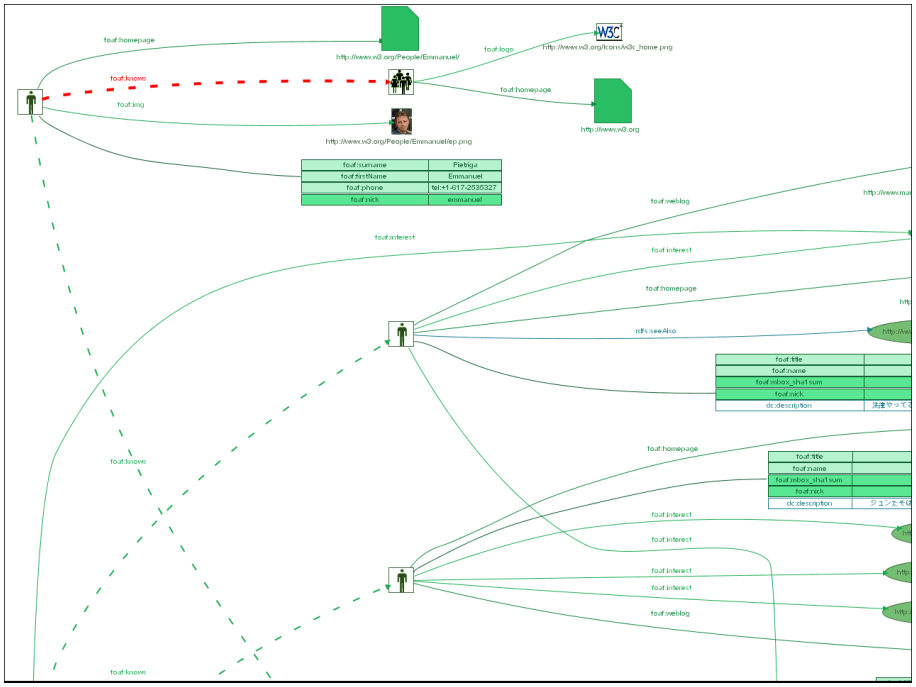

Figure 32: Part of an RDF document with GSS styling

formulation support; however, the availability of different style sheets can make IsaViz a flexible tool for generating representations targeted at both end-users and knowledge engineers (as well as, obviously, ontology developers).

4.3 Feature summary

In the previous section we have conducted a survey on some of the most widespread tools implementing some more or less sophisticated techniques for the visualization and exploration of ontological knowledge bases, and we have discussed them with reference to the evaluation criteria defined in subsection 4.1. Table 1 summarizes the characteristics of such tools.

Table 1: Schematic classification of the surveyed tools

Tool	*Config.*	*Multi metaphor*	*Orthog.*	*Uniform*	*Semantic browsing*	*Query formulation*
Ontolingua	no	yes	no	yes	yes	no
Ontosaurus	no	yes	no	yes	yes	no
Protégé	no	yes	no	no	yes	yes
OntoViz	partly	no	no	no	no	poor
OWLViz	no	no	no	no	no	poor
TGVizTab	yes	no	no	no	yes	poor
Jambalaya	yes	no	yes	no	yes	yes
WebOnto	no	no	no	no	yes	poor
Ontorama	no	no	no	no	no	poor
KAON	no	no	no	no	yes	yes
Spectacle	no	yes	no	no	yes	yes
IsaViz	yes	yes	yes	no	no	no

5 Conclusions

The analysis of the main query languages available for XML and the Semantic Web indicates that the majority of them could act as a good basis for supporting the querying needs of the Wisdom project. In particular, XQuery is the main candidate to be chosen as the Wisdom query language. Indeed, the high flexibility of its constructs, perhaps enriched with some of the constructs of the languages specifically designed for the semantic web, well satisfy the needs of the project for flexible information search in a heterogeneous/distributed ontologies scenario.

As to query rewriting approaches, and independently from the query language that will be eventually chosen, we have seen that many reformulation approaches exist in the literature in a standard relational scenario, which is indeed too different from the one of the Wisdom project in order to consider these approaches as feasible. However, some structural query rewriting techniques were also proposed and developed for systems managing heterogeneous XML document bases and in peer data management systems involving ontology data. Such techniques have thus been analyzed more in depth and they could be fruitfully adapted to the complex architecture of Wisdom.

Finally, in the knowledge presentation field, we have inspected and presented several visual languages, user interaction paradigms and tools. From this analysis the consideration that the intuitiveness and friendliness goals still seem to be far from being reached emerges; however, we hope that this critical discussion of the weaknesses of the available approaches will facilitate the hard task of defining in Wisdom novel formalisms and user interaction paradigms overcoming them.

References

[1] GraphViz - graph visualization software. [Online]. Available at http://www.graphviz.org/About.php.

[2] OntoViz tab: Visualizing Protégé ontologies. [Online]. Available at http://protege.stanford.edu/plugins/ontoviz/ontoviz.html.

[3] OWLViz - making OWL easier. [Online]. Available at http://www.co-ode.org/downloads/owlviz/co-ode-index.php.

[4] Protégé ontology editor and knowledge acquisition system. [Online]. Available at http://protege.stanford.edu/.

[5] Protégé OWL plugin. [Online]. Available at http://protege.stanford.edu/plugins/owl/.

[6] TGVizTab - a TouchGraph visualization tab for Protégé 2000. http://www.ecs.soton.ac.uk/ ha/TGVizTab/TGVizTab.htm.

[7] Touchgraph development. [Online]. Available at http://touchgraph.sourceforge.net/.

[8] Eclipse platform technical overview. Technical report, Object Technology International, Inc., February 2001. Available at http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[9] S. Alexaki, G. Karvounarakis, V. Christophides, D. Plexousakis, and K. Tolle. The ICS FORTH RDFSuite: Managing Voluminous RDF Description Bases. In *Proc. of the 2nd International Workshop on the Semantic Web*, 2001.

[10] Sean Bechhofer, Ian Horrocks, Carole Goble, and Robert Stevens. OilEd: a reason-able ontology editor for the semantic web. In *Proceedings of KI2001, Joint German/Austrian*

conference on Artificial Intelligence, number 2174 in Lecture Notes in Computer Science, pages 396–408, Vienna, September 2001. Springer-Verlag.

[11] David Beckett. Design and implementation of the redland RDF application framework. WWW10 Presentation, May 2001.

[12] A. Berglund, S. Boag (XSL WG), D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0. W3C Working Draft (4 April 2005), 2005.

[13] François Boutin and Mountaz Hascoët. Focus dependent multi-level graph clustering. In *Proceedings of the working conference on Advanced visual interfaces*, pages 167 – 170, May 2004.

[14] T. Bray, Jean Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0. W3C Recommendation (4 February 2004), 2004.

[15] D. Brickley and R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation (10 February 2004), 2004.

[16] Jeen Broekstra and Arjohn Kampman. Query language definition. On-To-Knowledge project deliverable, March 2001. [Online]. Available at http://sesame.administrator.nl/doc/del9.pdf.

[17] R. G. G. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez. *The Object Database Standard ODMG 3.0*. Morgan Kaufmann, 2000.

[18] D. Chamberlin. XQuery: An XML query language. *IBM Systems Journal*, 41(4), 2002.

[19] S. Cluet, P. Veltri, and D. Vodislav. Views in a large scale XML repository. In *Proc. of the 27th VLDB*, 2001.

[20] A. Deutsch and V. Tannen. A system for publishing xml from mixed and redundant storage. In *Proc. of the 29th VLDB*, 2003.

[21] John Domingue. Tadzebao and WebOnto: Discussing, browsing, and editing ontologies on the web. In B. Gaines and M. Musen (Eds), editors, *Proceedings of the 11^{th} Knowledge Acquisition for Knowledge-Based Systems Workshop*. SRDG Publications, 18-23 April 1998.

[22] John Domingue, Enrico Motta, and Oscar Corcho Garcia. *Knowledge Modelling in WebOnto and OCML: A User Guide*. Knowledge Media Institute, Milton Keynes, MK7 6AA, UK, 1999.

[23] Peter Eades. A heuristic for graph drawing. In *Congressus Numerantium*, pages 149–160, 1984.

[24] Neil A. Ernst, Margaret-Anne Storey, and Mark Musen. Addressing cognitive issues in knowledge engineering with jambalaya. *Workshop on Visualization in Knowledge Engineering, KCAP 03*, October 2003.

[25] Neil A. Ernst and Margaret-Anne D. Storey. A preliminary analysis of visualization requirements in knowledge engineering tools. Technical report, CHISEL Technical Report, University of Victoria, August 2003.

[26] A. Malhotra et al. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft (4 April 2005), 2005.

[27] M. Fernandez et al. XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft (4 April 2005), 2005.

[28] S. Agarwal et al. Semantic Methods and Tools for Information Portals. *GI Jahrestagung*, 1, 2003.

[29] S. Boag et al. XQuery 1.0: An XML Query Language. W3C Working Draft (4 April 2005), 2005.

[30] Adam Farquhar, Richard Fikes, Wanda Pratt, and James Rice. Collaborative ontology construction for information integration. Knowledge Systems Laboratory Department of Computer Science, August 1995.

[31] R. Fikes, P. Hayes, and I. Horrocks. OWL-QL: A Language for Deductive Query Answering on the Semantic Web. KSL Technical Report 03-14, 2003.

[32] Christiaan Fluit, Marta Sabou, and Frank van Harmelen. Ontology-based information visualisation. Springer Verlag, 2002.

[33] George W. Furnas. Generalized fisheye views. *Human Factors in computing systems, CHI '86 conference proceedings, ACM*, pages 16–23, 1986.

[34] J. Caroll G. Klyne. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation (10 February 2004), 2004.

[35] H. Garcia-Molina, S. Melnik, and E. Rahm. A Versatile Graph Matching Algorithm and its Application to Schema Matching. In *Proc. of the 18th ICDE*, 2002.

[36] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate XML joins. In *Proc. of ACM SIGMOD*, 2002.

[37] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *Proc. of the 19th ICDE*, 2003.

[38] Siegfried Handschuh, Alexander Maedche, Ljiljana Stojanovic, and Raphael Volz. KAON - the KArlsruhe ONtology and semantic web infrastructure.

[39] Ian Horrocks. *Optimising Tableaux Decision Procedures for Description Logics*. PhD thesis, University of Manchester, 1997.

[40] G. Karvounarakis, A. Magganaraki, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, and K. Tolle. Querying the Semantic Web with RQL. *Computer Networks*, 42(5), 2003.

[41] Graham Kline and Jeremy J. Carrol. Resource description framework (RDF: Concepts and abstract syntax. [Online]. Available at http://www.w3.org/TR/rdf-concepts/, February 2004. W3C Recommendation.

[42] P. Lehti, N. Shreshta, and S. Hollfelder. The Semantic Web Query Language SWQL. IPSI Franhofer Working Draft, 2003.

[43] Alexander Maedche, Boris Motik, and Raphael Volz. KAON - a framework for semantics based e-services. Institute AIFB, University of Karlsruhe.

[44] A. Magkanaraki, V. Tannen, V. Christophides, and D. Plexousakis. Viewing the Semantic Web through RVL Lenses. In *Proc. of the International Semantic Web Conference*, 2003.

[45] F. Mandreoli, R. Martoglia, and P. Tiberio. Approximate Query Answering for a Heterogeneous XML Document Base. In *Proc. of the 5th WISE Conference*, 2004.

[46] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, 2004.

[47] Jeff Michaud and Margaret-Anne Storey. The role of knowledge in software customization. *15th Internation Conference on Software Engineering and Knowledge Engineering (SEKE03)*, 2003.

[48] L. Miller. Inkling: RDF query using squishql. [Online]. Available at http://swordfish.rdfweb.org/rdfquery.

[49] Paul Mutton and Peter Rogers. Spring embedder preprocessing for www visualization. In *Proceedings Information Visualization 2002. IVS, IEEE*, pages 744–749, July 2002.

[50] B. Ooi, Y. Shu, and K. L. Tan. Relational data sharing in peer-based data management systems. *SIGMOD Record*, 23(3), 2003.

[51] G. Papamarkos, A. Poulovassilis, and P. T. Wood. RDFTL: An Event-Condition-Action language for RDF. In *Proc. of the 3rd International Workshop on Web Dynamics*, 2004.

[52] P.Zezula, F. Mandreoli, and R. Martoglia. Unordered XML Pattern Matching with Tree Signatures. In *Proc. of the 12th Convegno su Sistemi Evoluti per Basi di Dati (SEBD 2004)*, 2004.

[53] Manojit Sarkar, Scott S. Snibbe, Oren J. Tversky, and Steven P. Reiss. Stretching the rubber sheet: A metaphor for viewing large layouts on small screens. In *Proceedings of ACM UIST '93*, pages 81–91. ACM Press, 1993.

[54] T. Schlieder and F. Naumann. Approximate tree embedding for querying XML data. In *Proc. of the ACM SIGIR Workshop On XML and Information Retrieval*, 2000.

[55] A. Seaborne. RDQL Tutorial for Jena. HP Labs, 2002.

[56] A. Seaborne. RDQL - A Query Language for RDF. HP Labs Submission to the W3C, 2004.

[57] D. Shasha, J.T.L.Wang, H. Shan, and K. Zhang. ATreeGrep: Approximate Searching in Unordered Trees. In *Proc. of the 14th International Conference on Scientific and Statistical Database Management*, 2002.

[58] Michael Sintek. OntoViz tab: Visualizing Protégé ontologies. [Online]. Available at http://protege.stanford.edu/plugins/ontoviz/ontoviz.html, 2003.

[59] Bill Swartout, Ramesh Patil, Kevin Knight, and Tom Russ. Toward distributed use of large-scale ontologies. In *Proceedings of the 10th Banff Knowledge Acquisition Workshop*, November 9-14 1996.

[60] I. Tatarinov and A. Halevy. Efficient Query Reformulation in Peer Data Management Systems. In *Proc. of ACM SIGMOD*, 2004.

[61] P. Zezula, G. Amato, F. Debole, and F. Rabitti. Tree Signatures for XML Querying and Navigation. In *Proc. of the XML Database Symposium*, 2003.

[62] P. Zezula, F. Mandreoli, and R. Martoglia. Tree Signatures and Unordered XML Pattern Matching. In *Proc. of the 30th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2004)*, 2004.

[63] K. Zhang, R. Statman, and D. Shasha. On the editing distance between unordered labeled trees. *Inf. Process. Lett.*, 42(3), 1992.