

ONTOXPL - EXPLORATION OF OWL ONTOLOGIES

YING LU

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

SEPTEMBER 2004

© YING LU, 2004



Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*

ISBN: 0-612-94748-3

Our file *Notre référence*

ISBN: 0-612-94748-3

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

OntoXpl - Exploration of OWL Ontologies

Ying Lu

To complement existing ontology editors, OntoXpl has been designed and implemented for searching and browsing knowledge bases in ontology files. The main motivation for OntoXpl is to provide a user-friendly, robust, and scalable ontology visualization and exploration tool for different users. The idea is to help users narrow down the knowledge base, for a given query, retrieve the relevant information, organize, and display the result in a manner convenient to users. The system relies on Racer's inference capabilities for offering users a better exploration mechanism.

OntoXpl is based on the tomcat architecture and is available on the web, through standard HTML browsers. OntoXpl is one of the few ontology exploration tools developed so far that is fully targeted to OWL.

At least three potential user groups are targeted by OntoXpl's design: (i) users with a limited background of ontologies and OWL; (ii) ontology developers who are OWL experts; (iii) users interested in understanding and reusing existing ontologies.

Acknowledgments

Great thanks must go to my supervisors, Dr. Haarslev and Dr. Shiri, for their knowledgeable input, guidance and encouragement throughout the duration of this research. It is they who had patiently lead me into this area. I really appreciate their sincere help.

Two years ago, I knew nothing about my research. It is really a challenge and hard work for me to model and implement the OntoXpl system. However, when I review what I have done, I found the whole procedure that I have experienced becomes an unforgettable treasure of my life.

I would also like to thank my parents, who have always encouraged me, believed in me, and supported me to overcome difficulties. Besides, I thank all my friends at Concordia University.

Table of Contents

List of Figures	vii
List of Tables	x
1. Introduction	1
1.1. <i>Web Ontology Language (OWL)</i>	1
1.2. <i>Concept Definitions in OWL</i>	3
1.3. <i>Role Definition in OWL</i>	5
1.4. <i>Individual Definitions in OWL</i>	9
2. Problems	10
2.1. <i>Ontology: An Introduction</i>	12
2.2. <i>Ontology design, revision, and understanding</i>	13
3. Ontology exploration system – OntoXpl Tool	14
3.1. <i>Main menu functions</i>	16
3.2. <i>Natural language explanation</i>	21
3.3. <i>Inferred information</i>	29
3.4. <i>Axiom browsing</i>	37
3.5. <i>Abox queries</i>	43
3.6. <i>Taxonomy</i>	53
3.7. <i>New Racer Query Language nRQL query interface</i>	57
4. Related work	61
4.1. <i>Protégé and OntoXpl</i>	62
4.2. <i>OntoEdit</i>	70
4.3. <i>KAON</i>	72
4.4. <i>Apollo</i>	73
4.5. <i>Summary</i>	75
4.6. <i>Interface for Knowledge Retrieval</i>	79

5. Design and implementation	81
5.1. <i>OntoXpl Architecture</i>	81
5.2. <i>Modules and relationship</i>	93
6. Evaluation	104
7. Conclusions and Future work	105
7.1. <i>Conclusions</i>	105
7.2. <i>Future work</i>	106
8. References	108
Appendix	111
A.1 <i>Example about loading OWL ontology into OntoXpl</i>	111
A.2 <i>Appendix 2 OntoXpl API and package information</i>	112

List of Figures

FIGURE 1.1: OWL DEFINITION FOR “ <i>DISNEY_CAT</i> ”	3
FIGURE 1.2: DL SYNTAX FOR “ <i>DISNEY_CAT</i> ”	4
FIGURE 1.3: OWL: EQUIVALENT CLASS EXAMPLE	5
FIGURE 1.4: A TRANSITIVE ROLE DEFINITION	5
FIGURE 1.5: INDIVIDUALS USING TRANSITIVE ROLE	6
FIGURE 1.6: THE INFERRED RESULT OF TRANSITIVE ROLE	6
FIGURE 1.7: A SYMMETRIC ROLE DEFINITION	7
FIGURE 1.8: INDIVIDUALS USING SYMMETRIC ROLE	7
FIGURE 1.9: THE INFERRED RESULT OF SYMMETRIC ROLE	7
FIGURE 1.10: INVERSE ROLE DEFINITION	8
FIGURE 1.11: INDIVIDUALS USING INVERSE ROLE	8
FIGURE 1.12: THE INFERRED RESULT OF INVERSE ROLE	8
FIGURE 1.13: INDIVIDUAL DEFINITION	9
FIGURE 3.1: ONTOXPL’S MAIN PAGE	17
FIGURE 3.2: ONTOLOGY LOADING PAGE	18
FIGURE 3.3: ONTOXPL’S MAIN PAGE WITH THE DISPLAYED ONTOLOGY INFORMATION	19
FIGURE 3.4: ZOOM OF THE UPPER THREE COMMAND MENUS (FROM LEFT TO RIGHT): FILE SELECTION, OWL / NATURAL LANGUAGE VIEWS, INFORMATION PAGE VIEWS	20
FIGURE 3.5: ZOOM OF THE MIDDLE TWO COMMAND MENUS (FROM LEFT TO RIGHT): EXPLORE CONCEPT/PROPERTY CHARACTERISTICS, SHOW CONCEPT/ROLE HIERARCHIES	20
FIGURE 3.6: ZOOM OF THE BOTTOM TWO COMMAND MENUS (FROM LEFT TO RIGHT): A-BOX COMMAND MENU, nRQL RACER QUERY LANGUAGE	21
FIGURE 3.7: VIEW FUNCTION: CONCEPTS, ROLES AND INDIVIDUALS	22
FIGURE 3.8: LIST CONCEPTS IN THE DOMAIN	23
FIGURE 3.9: OWL SPECIFICATION OF CLASS “ <i>CARTOON_CAT</i> ”	24
FIGURE 3.10: “NATURAL LANGUAGE” DESCRIPTION OF CLASS “ <i>CARTOON_CAT</i> ”	24
FIGURE 3.11: “ <i>CARTOON_CAT</i> ” OWL DEFINITION AND NL EXPLANATION MAIN PAGE	25
FIGURE 3.12: “ <i>CARTOON_MOUSE</i> ” OWL DEFINITION AND NL EXPLANATION	26
FIGURE 3.13: “ <i>CARTOON_MOUSE</i> ” NL-EXPLANATION	26
FIGURE 3.14: “ <i>CARTOON_MOUSE</i> ” DL-DEFINITION	26
FIGURE 3.15: EMPIRICAL FACT EXAMPLE “FEATURE”	27
FIGURE 3.16: “ <i>IN_SAME_CARTOON_SERIES</i> ” ROLE NL-EXPLANATION MAIN PAGE	27
FIGURE 3.17 NL-EXPLANATION FOR ROLE “ <i>IN_SAME_CARTOON_SERIES</i> ”	28
FIGURE 3.18: “ <i>MICKEY</i> ” INDIVIDUAL NL-EXPLANATION MAIN PAGE	28
FIGURE 3.19: NL-EXPLANATION FOR INSTANCE “ <i>MICKEY</i> ”	28
FIGURE 3.20: “ <i>CARTOON_MOUSE</i> ” CONCEPT INFORMATION PAGE	30
FIGURE 3.21: “ <i>VOICED_BY</i> ” ROLE INFORMATION PAGE	32
FIGURE 3.22: “ <i>IN_SAME_SERIES</i> ” INDIVIDUAL INFORMATION PAGE	32
FIGURE 3.23: INDIVIDUAL INFORMATION MAIN PAGE	33
FIGURE 3.24: “ <i>MINNIE</i> ” INDIVIDUAL INFORMATION PAGE	34
FIGURE 3.25: CASE STUDY EXAMPLE FOR “INDIVIDUAL INFERENCE”	36
FIGURE 3.26: CASE STUDY RESULT ABOUT INDIVIDUAL INFERENCE	36
FIGURE 3.27: ONTOXPL INSTANCE-INFO RESULT FOR “ <i>INDI21</i> ”	36

FIGURE 3.28: EQUIVALENT-CONCEPT MAIN PAGE	37
FIGURE 3.29: DISJOINT CONCEPT BY PAIR	39
FIGURE 3.30: DISJOINT CONCEPT NAME INPUT	39
FIGURE 3.31: SEARCH CONCEPT WITH ONE WORD “ <i>CARTOON</i> ”	40
FIGURE 3.32: DISJOINT RESULTS -“ <i>CARTOON_CAT</i> ”	40
FIGURE 3.33: DISJOINT RESULT FOR “ <i>MICKEY_MOUSE</i> ”	41
FIGURE 3.34: SYMMETRIC ROLES IN A SUMMARY REPORT	42
FIGURE 3.35: INVERSE ROLES IN A SUMMARY REPORT	42
FIGURE 3.36: TRANSITIVE ROLE IN A SUMMARY REPORT	43
FIGURE 3.37: “INDIVIDUALS->ROLES->INDIVIDUALS”: ALL INSTANCES RELATED TO OTHER INSTANCES	44
FIGURE 3.38: IN_SAME_CARTOON_SERIES: (<i>GOOFY, MICKEY</i>)(<i>MICKEY, MINNIE</i>)(<i>MINNIE, MILLICENT</i>)(<i>MINNIE, MELODY</i>)(<i>PLUTO, GOOFY</i>)	45
FIGURE 3.39: ROLES USED BY AND INSTANCES RELATED TO “ <i>GOOFY</i> ”	45
FIGURE 3.40: “ROLES -> RELATED INDIVIDUAL PAIRS”: ALL ROLES USED BY INSTANCES	46
FIGURE 3.41: “ <i>FIRST_APPEARANCE</i> ” TEMPLATE II RESULTS	47
FIGURE 3.42: ROLE(<i>INDIX, INDIY</i>) IS USED ONLY ONCE	48
FIGURE 3.43: ROLES USED ONLY ONCE BY INSTANCES IN THE DOMAIN	48
FIGURE 3.44: INPUT INDIVIDUAL NAME	49
FIGURE 3.45: INDIVIDUAL NAME SEARCH RESULTS	49
FIGURE 3.46: SEARCH BY INDIVIDUAL NAME RESULT	50
FIGURE 3.47: INDIVIDUAL USING DATATYPE ROLE SEARCH RESULT	51
FIGURE 3.48: BROWSING ALL INSTANCES	52
FIGURE 3.49: SPACE TREE FORMAT OUTPUT FOR INSTANCE “ <i>MICKEY</i> ”	53
FIGURE 3.50: USER’S CONCEPT DEFINITION	54
FIGURE 3.51: CONCEPT HIERARCHY INFERENCE RESULTS	54
FIGURE 3.52: SPACETREE CONCEPT HIERARCHY RESULTS	55
FIGURE 3.53: SPACETREE ROLE HIERARCHY RESULTS	55
FIGURE 3.54: HTML/SCRIPT CONCEPT AND ROLE HIERARCHY	56
FIGURE 3.55: HTML/SCRIPT CONCEPT AND ROLE HIERARCHY	57
FIGURE 3.56: SIMPLE ABOX QUERY THROUGH NRQL INTERFACE	58
FIGURE 3.57: QUERY “ <i>DISNEY_MOUSE</i> ”, WHO HAS NIECES, AND IS A FRIEND OF <i>MICKEY</i>	59
FIGURE 3.58: EXAMPLE NRQL QUERY AND ITS RESULT	59
FIGURE 3.59: QUERY RESULTS FOR SCENARIO TWO	60
FIGURE 3.60: ONTOXPL ABOX COMPLEX QUERY RESULTS IN A LISP-LIKE NOTATION	60
FIGURE 4.1: CONCEPT NAME BEGINS WITH A NUMBER	63
FIGURE 4.2: “ <i>TEST.OWL</i> ” DESIGNED BY PROTÉGÉ AND GUI RESULTS	64
FIGURE 4.3: MOST SPECIFIC TYPES FOR “ <i>INDI1</i> ” AND “ <i>INDI2</i> ”	64
FIGURE 4.4: REFERRED RESULTS BY ONTOXPL	65
FIGURE 4.5: ABOX SPACE-TREE GENERATION RESULT	68
FIGURE 4.6: ONTOEDIT KNOWLEDGE BASE VISUALIZATION	71
FIGURE 4.7: ONTOEDIT GUI	72
FIGURE 4.8: KAON MAIN PAGE	73
FIGURE 4.9: APOLLO HIERARCHY INFORMATION	74
FIGURE 6.1: THE ONTOXPL SYSTEM STRUCTURE	82

FIGURE 6.2: JENA PARSING GRAPH RESULT – FROM HTTP://JENA.SOURCEFORGE.NET/TUTORIAL/RDF_API/FIGURES/FIG2.PNG	86
FIGURE 6.3: JENA N3 RDF TRIPLE TRANSLATION RESULTS	86
FIGURE 6.4: OWL EXAMPLE FOR JENA PARSER	87
FIGURE 6.5: JENA RDF RESULT	87
FIGURE 6.6: JENA N3 RESULT	88
FIGURE 6.7: SAX APIS	90
FIGURE 6.8: NL PACKAGE API INFO	91
FIGURE 6.9: NL PARSING ALGORITHM	91
FIGURE 6.10: DATABASE OPERATION ARCHITECTURE	92
FIGURE 6.11: SERVLET AND REASONING API MODULES	94
FIGURE 6.12: NATURAL LANGUAGE DESCRIPTION FUNCTION SCHEMA OVERVIEW	95
FIGURE 6.13: OWL SOURCE DEFINITION GENERATION ALGORITHM	96
FIGURE 6.14: NATURAL LANGUAGE TRANSLATION FLOWCHART	97
FIGURE 6.15: OWL DEFINITION FOR EXAMPLE CONCEPT “XI”	97
FIGURE 6.16: ALGORITHM FOR REASONING RESULTS BASED ON RACER	98
FIGURE 6.17: UML PACKAGE INFORMATION FOR “INFORMATION.SERVLET” PACKAGE	99
FIGURE 6.18: UML PACKAGE INFORMATION FOR REASONING.JAVA CLASS	100
FIGURE 6.19: SPACETREE OUTPUT GENERATION	101
FIGURE 6.20: EXAMPLE ONTOLOGY SOURCE FILE	102
FIGURE 6.21: SPACETREE SYNTAX OUTPUT FILE	102
FIGURE 6.22: SPACETREE CONCEPT HIERARCHY OUTPUT GENERATION ALGORITHM	103
FIGURE 6.23: SPACETREE INDIVIDUAL RELATIONSHIP OUTPUT GENERATION ALGORITHM	103
FIGURE A2.1: RACERPART PACKAGE INFO	112
FIGURE A2.2: REASONING RACERPART	113
FIGURE A2.3: NATURAL LANGUAGE FUNCTIONS	113
FIGURE A2.4: NL CONCEPT PARSING CLASS	114
FIGURE A2.5: NL ROLE PARSING CLASS	115
FIGURE A2.6: NL INSTANCE PARSING CLASS	116
FIGURE A2.7: DIG TRANSLATION FUNCTION	117
FIGURE A2.8: SERVLET PACKAGE INFORMATION	118
FIGURE A2.9: SERVLET STRUCTURE	119

List of Tables

TABLE 4.1: COMPARISON CRITERIA	76
TABLE 4.2: MAIN FEATURES COMPARISON OF APOLLO, OILED, ONTOEDIT, PROTÉGÉ, KOAN, AND ONTOXPL	78

1. Introduction

“The Web Ontology Language OWL is a semantic markup language for publishing and sharing ontologies on the World Wide Web. OWL is developed as a vocabulary extension of RDF (Resource Description Framework) and is derived from the DAML+OIL Web Ontology Language...” [8].

1.1. Web Ontology Language (OWL)

Description logic systems play an ever-growing role in knowledge representation and reasoning. In particular, the semantic web initiative is based on description logics (DLs) and poses important challenges for development of systems. Recently, Web Ontology Language (OWL) has been proposed as a standard for the semantic web. OWL is based on two other standards: Resource Description Format (RDF) [16] and its corresponding “vocabulary language”, RDF Schema (RDFS) [4]. In recent research efforts, these languages are mainly considered as ontology representation languages, used for defining classes of so-called *abstract objects*. Nowadays, many applications would start using the RDF part of OWL for representing information about specific abstract objects in a certain domain. Graphical editors such as OilEd [18] or Protégé [15] support this way of using OWL quite well [27]. As a language for defining web ontology and their associated knowledge bases, OWL is a revision of DAML+OIL [19] ontology language, which satisfies all requirements of ontology language by providing both syntactic and semantic

interoperability. Compared with RDF and RDFS, OWL provides constructors that are more expressive such as equivalence, disjoint, necessary and sufficient conditions and instance definition.

In order to help readers better understand the remaining chapters, we provide an OWL overview in this section. As a mesh of information linked together to help machines read and analyze source knowledge, the Semantic Web [21] envisions that machines, instead of human beings, could do more of the hard work. Ontology, constituted by a specific vocabulary in a certain domain, provides this semantic interoperability by defining data models in terms of concepts, roles, and instances. In order to add a machine-tractable logic layer to the “natural language” web of HTML, we need a well-understood and formally specified language, which should also be relatively easy to use and amenable to machine processing. At present, OWL Web Ontology Language is the best choice, which facilitates, by and large, greater machine interpretability of web content than that supported by XML, RDF, and RDF Schema (RDF-S) due to the additional vocabulary along with a formal semantics. OWL has three sublanguages: OWL Lite, OWL DL, and OWL Full [5]. In the following sections, we are going to use a vivid ontology example of cartoon stars to show how OWL corresponds to description logics, and how one can define classes, relationships, and individuals in order to capture the semantics of data.

1.2. Concept Definitions in OWL

We assume “*Disney_cat*”, which is a subclass of the concept “*Disney_cartoon_star*”, is defined as “a subclass of *Cartoon_cat* and is produced only by *Disney*”. Figure 1.1 illustrates the OWL presentation for the definition of “*Disney_cat*.”

```
<owl:Class rdf:ID="Disney_cat">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="collection">
        <owl:Class rdf:about="#Cartoon_cat"/>
        <owl:Restriction>
          <owl:allValuesFrom>
            <owl:Class>
              <owl:oneOf rdf:parseType="collection">
                <Producer rdf:ID="Disney"/>
              </owl:oneOf>
            </owl:Class>
          </owl:allValuesFrom>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty>
            <owl:ObjectProperty rdf:about="#Produced_by"/>
          </owl:onProperty>
          <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Disney_cartoon_star"/>
  </rdfs:subClassOf>
</owl:Class>
```

Figure 1.1: OWL definition for “*Disney_cat*”

Note that one may use `<owl:Class rdf:ID="ConceptName">` to define a named class. In the example, “*Disney_cat*” is defined as `<owl:Class rdf:ID="Disney_cat">`. OWL also supports the basic set operations, namely union, intersection, and complement, which are named *owl:unionOf*, *owl:intersectionOf*, and *owl:complementOf*, respectively. Besides, *owl:minCardinality* and *owl:maxCardinality* are used to specify the lower and upper bounds. In the example, *Disney_cat* is defined as the intersection of class *Cartoon_cat*

and two anonymous classes. Comparing with named classes, anonymous classes are the classes defined without names such as `<owl:Class>` as shown in Figure 1.1. With *allValuesFrom* and *minCardinality* restrictions on the *produced_by* property, two anonymous classes have been generated: one is defined as a concept whose value for *produced_by* property must be *Disney*; the other is defined as a concept, which has at least one value for *produced_by* property. Now with the intersection of these two anonymous classes and the previously defined *Cartoon_cat* class, we can express exactly “produced by Disney”. Figure 1.2 displays the DL syntax of this example.

DL syntax
$\text{Disney_cat} \equiv \text{Cartoon_cat} \sqcap \forall \text{Produced_by}.\{\text{Disney}\} \sqcap \geq 1 \text{Produced_by}$
$\text{Disney_cat} \sqsubseteq \text{Disney_cartoon_star}$

Figure 1.2: DL Syntax for “*Disney_cat*”

owl:equivalentClass provides a way to define a necessary and sufficient condition. The meaning of such a class axiom is that the two class descriptions involved have the same class extension (i.e., both class extensions contain exactly the same set of individuals). In its simplest form, an equivalent Class axiom states the equivalence (in terms of their class extension) of two named classes. An example about the “*Cartoon_star*” definition is shown in Figure 1.3.

```

<owl:Class rdf:ID="Cartoon_star">
  <owl:equivalentClass>
    <owl:Class rdf:about="#Animation_star"/>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="Animation_star">
  <owl:equivalentClass rdf:resource="#Cartoon_star"/>
</owl:Class>

```

Figure 1.3: owl:equivalentClass example

1.3. Role Definition in OWL

OWL also provides property characteristics such as transitive, symmetric, inverseOf, functional, and inverse functional expressions. Now we will give three examples to explain *owl:TransitiveProperty*, *owl:SymmetricProperty*, and *owl:inverseOf*. When one defines a property P to be a transitive property, this means that if a pair (x, y) is an instance of P, and the pair (y, z) is instance of P, then we can infer that the pair (x, z) is also an instance of P. For instance, we define a transitive role “*In_same_cartoon_series*” in Figure 1.4.

```

<owl:TransitiveProperty rdf:ID="In_same_cartoon_series">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="collection">
        <owl:Class rdf:about="#Cartoon_star"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <rdfs:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
  <rdfs:range rdf:resource="#Cartoon_star"/>
  <rdfs:type rdf:resource="http://www.w3.org/2002/07/owl#SymmetricProperty"/>
  <rdfs:subPropertyOf>
    <owl:SymmetricProperty rdf:about="#In_same_series"/>
  </rdfs:subPropertyOf>
</owl:TransitiveProperty>

```

Figure 1.4: A transitive role definition


```

<Disney_dog rdf:ID="Goofy">
  <In_same_cartoon_series rdf:resource="Donald_Duck"/>
</Disney_dog>

<Disney_duck rdf:ID="Donald_Duck">
  <In_same_cartoon_series>
    <Disney_micky_mouse rdf:ID="Mickey"/>
  </In_same_cartoon_series>
</Disney_duck>

```

Figure 1.5: individuals using transitive role

In Figure 1.5, two individuals “Goofy” and “Donald_Duck” are defined by a transitive role “In_same_cartoon_series”. The “Cartoon_star” ontology also defines that “Goofy and Donald_Duck are in the same cartoon series”, “Donald_Duck and Mickey are in the same cartoon series”, that is, we have “In_same_cartoon_series(Goofy, Donald_Duck)”, and “In_same_cartoon_series(Donald_Duck, Mickey).” After connecting with the OWL reasoner Racer [24], we can get *In_same_cartoon_series(Goofy, Mickey)*, shown in Figure 1.6.

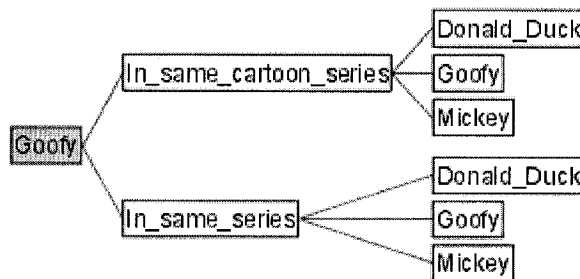


Figure 1.6: The inferred result of transitive role

A symmetric property is a property for which it holds that if the pair (x, y) is an instance of P, then the pair (y, x) is also an instance of P. Syntactically, a property is defined as symmetric by making it an instance of the built-in OWL class owl:SymmetricProperty, a

subclass of owl:ObjectProperty. In the cartoon star example, we define “*Is_friend_of*” as a symmetric role in Figure 1.7.

```
<owl:symmetricProperty rdf:ID="Is_friend_of">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
</owl:symmetricProperty>
```

Figure 1.7: A symmetric role definition

In Figure 1.8, we have defined *Is_friend_of*(Odie, Garfield), which means *Is_friend_of*(Garfield, Odie) as shown in Figure 1.9.

```
<Disney_dog rdf:ID="Odie">
  <Is_friend_of rdf:resource="#Garfield"/>
  <In_same_cartoon_series rdf:resource="#Garfield"/>
</Disney_dog>
```

Figure 1.8: Individuals using symmetric role

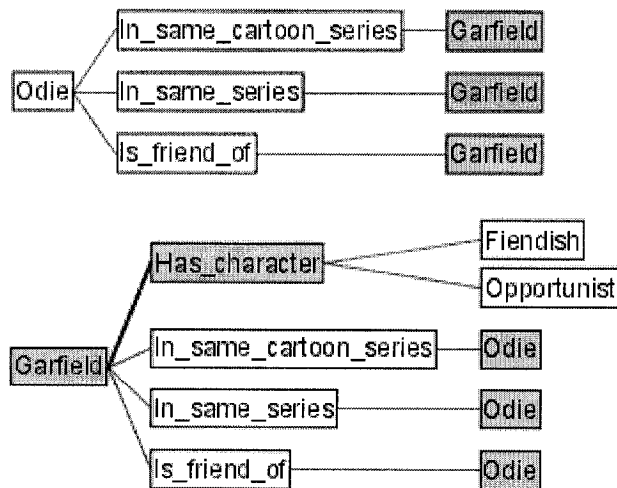


Figure 1.9: The inferred result of symmetric role

One property can be stated to be the inverse of another property. If property P1 is stated to be the inverse of property P2, then if X is related to Y by P2, then Y is related to X by P1. For example, if “Eat” is the inverse property of “Eaten_by” (in Figure 1.10) and “tom Eat jerry” (in Figure 1.11), then an OWL reasoner can deduce that “jerry Eaten_by tom” (in Figure 1.12).

```
<owl:ObjectProperty rdf:ID="Eaten_by">  
  <owl:inverseOf rdf:resource="#Eat"/>  
</owl:ObjectProperty>
```

Figure 1.10: Inverse role definition

```
<Disney_cat rdf:ID="Tom">  
  <Eat rdf:resource="#Jerry_Mouse"/>  
</Disney_cat>
```

Figure 1.11: individuals using inverse role

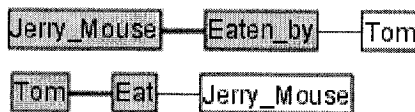


Figure 1.12: The inferred result of inverse role

1.4. Individual Definitions in OWL

In addition to property and concept constructors, individuals also play an important role in OWL. Normally defined with individual axioms (called “facts” as well), an individual is minimally introduced by declaring it to be a member of a class. There are two types of facts:

1. Facts about class membership and property values of individuals
2. Facts about individual identity

Many facts typically are statements indicating class membership of individuals and property values of individuals. As an example, consider the set of statements in Figure 1.13 about an instance of the class “*Cartoon_cat*”:

```
<Disney_cat rdf:ID="Tom">
  <In_film rdf:resource="#Tom_and_Jerry"/>
  <Eat rdf:resource="#Jerry_Mouse"/>
</Disney_cat>
```

Figure 1.13: Individual definition

2. Problems

According to Merriam-Webster dictionary, Ontology is “a branch of metaphysics concerned with the nature and relations of being; a particular theory about the nature of being or the kinds of existents”, [<http://www.m-w.com/>].

Ontologies play a key role in knowledge discovery and knowledge sharing because of their logically syntactic structure and inference functions. Thus, designing a good ontology that represents the domain knowledge correctly and accurately becomes an important task. Currently, OWL is the most expressive and popular ontology language used by ontology and ontology services design. To facilitate ontology design, there are a number of ontology editors such as Protégé, OilEd, OntoEdit [28] and so on. Unfortunately, only a limited number of them support OWL well. For example, an ontology designed by one tool might not be opened or loaded successfully by other tools. Details about these tools and their comparison are provided in Chapter 4.

In this research, ontology design is not the main topic. Our approach is to develop a tool to explore an ontology, provide effectively organized information, and illustrate the knowledge base of the domain. The information includes not only the explicitly defined objects, but also the implicit information retrieved by communicating with an OWL reasoner such as Racer. As a result, this tool is able to ease users' understanding an ontology by quickly providing them with a global picture of the ontology in a user-friendly way. Users might be divided into three main groups: (1) people who have a

limited or little background of ontologies, (2) ontology experts or ontology developers, and (3) people who want to reuse ontologies designed by others. Ontology design is a time-consuming task; ontology developers can add more and more information and change their design periodically. As a result, an ontology file can grow into a very large and complex knowledge base to a point when it is even hard for the designers themselves to remember all the concepts, roles, instances, axioms, and the intrinsic relationship among those objects in their domain. A tool, which can help users explore an ontology, provides a fast overview of the design, and eases user acquisition of information according to complex query requirements, is demanded by both ontology developers and ontology users. Besides, newbies of ontology design have to learn how to organize metadata in an ontological way, emphasizing on concepts, roles, and instances. For ontology users, they are interested in whether the ontologies at hand are useful for them and for decisions on whether they can use them fully, partly, or not use it at all. Under this situation, they need a tool to help them catch the primary or highlighted information fast. In addition, when users are interested about a specific question or problem, only the related information should be displayed to them. To answer the questions mentioned above, we developed the “*Ontology Exploration tool – OntoXpl*”, which supports the ontology language OWL. OntoXpl can help ontology developers, ontology newbies, ontology reusers, and interested people to understand an ontology quickly and explore the ontology conveniently, by providing efficiently and effectively organized information with an intuitive and user-friendly interface. We will elaborate more on these features in Chapter 3.

2.1. Ontology: An Introduction

In order to support information search, we need information to be self-describing. Metadata provides a way to make data understandable and shareable for users over time [11]. Metadata such as the Dublin Core [7] attempts to define data by agreeing on the meaning of a set of annotation tags, but it might not be flexible enough and might limit the number of items that can be expressed. In addition, due to different understanding based on different cultural, social backgrounds, and knowledge, it is hard to set up a commonly understandable terminology or vocabulary. “Ontologies seek to provide a definitive and exhaustive classification of entities in all spheres of existence.” [3]. Derived from philosophy, in computer science “An ontology is an explicit specification of an abstract, simplified view of a world we desire to represent.” [9]. An ontology uses a vocabulary of formally specified terms to describe the world – providing a shared understanding of a domain of interest or knowledge. An ontology becomes the basic level of a knowledge representation scheme, which standardizes terminologies in form of taxonomies: classes, relations, and axioms. In addition, ontologies help make the implicit assumptions explicit, specify the unambiguous definition of terms, add semantic meaning of information, and support automatic reasoning based on defined rules. In order to realize these functions, various ontology languages have been developed: XML based SHOE [12], RDF, RDF Schema, DAML+OIL to the most popular used OWL. Adding a semantic layer – the machine-readable metadata, to the current web, ontologies work as a media helping “Agents” to communicate among each other to derive information.

2.2. Ontology design, revision, and understanding

As mentioned in section 2.1, ontology languages provide a way to share and reuse knowledge about phenomena in the world of interest. Designing a substantial ontology is not trivial, and designing ontologies in a way that provides relevance and value to a broad audience is even more challenging. In order to design and maintain high quality ontologies, there are four rules of thumb. First, give all named classes, roles, and instances a meaningful name. Second, talk or consult with people having substantial domain knowledge, the domain experts if possible, about what are to be designed by users, and try to specify correct and as clear as possible definitions. Third, avoid unnecessary synonyms and reduce redundancy. Last but not least, provide detailed descriptions for axioms. In building and applying an ontology, it is required to clarify two more important things: one is an ontology itself, which specifies concepts used in a domain of endeavor, concepts whose existence and relationships are true by definition or convention; another are empirical facts about these concepts and relationships, which are not part of the ontology, although they are structured by it. These empirical facts are subject to context, observation, testing, evaluation, or modification [2]. We will talk about empirical facts and give examples for them in section 3.2.

3. Ontology exploration system – OntoXpl Tool

State-of-the-art description logic (DL) inference systems such as Racer allow for interpreting OWL ontology documents as T-boxes and A-boxes [25]. Racer accepts the OWL DL subset (with the additional restriction of approximated reasoning for so-called nominals and no full number restrictions for datatype properties). Descriptions of individuals are represented as A-boxes by the Racer System. Viewing the RDF part of OWL DL documents as A-boxes provides for query languages supported by DL systems. Furthermore, graphical interfaces for description logic inference systems can be used to inspect OWL ontologies.

It is important to provide user interfaces for practical work with description logic inference systems. An increasing number of graphical interfaces are available for existing DL systems. One class of interfaces consists of ontology editors such as OilEd and Protégé. With these editors, ontologies can be interactively built and stored, for example, as OWL documents. In addition, the editors can be used to develop RDF documents for describing information about individuals with respect to OWL ontologies. Applications using these OWL documents require an inference engine that supports reasoning about individuals. Indeed, OilEd and Protégé can be configured to use Racer as an inference engine for classifying ontologies and for answering queries about individuals.

The second class of interfaces offers browsing and visualization capabilities. Rice [17] supports the input of textual queries and displays the concept/class hierarchy of Tboxes

as outline views as well as the relational structure of A-boxes as directed graphs. The outline view of classes is usually also supported by ontology editors but Rice additionally supports the visualization of A-boxes. Other OWL/RDF visualization tools or editors with visualization capabilities are, e.g., KAON [6], OntoEdit [28], and OntoTrack [22].

The design of OntoXpl is influenced by OWL (and its foundation on DLs). Therefore, it focuses on the three main language elements of OWL: classes/concepts, roles/properties, and nominals/individuals. In order to understand OWL ontologies quickly, we provide three primary kinds of information: concepts, roles, and instances, and organize them in a closely packed way.

The OWL ontology explorer OntoXpl which we have developed is intended to complement existing ontology editors and visualization tools. Now we will begin to introduce the main functions of the system and illustrate how functions presented by OntoXpl help users achieve their goals. The capabilities of OntoXpl are best explored interactively. However, in this chapter we try to illustrate some of its main features. Let us assume that OntoXpl is used to explore an ontology file called “*Cartoon_star.owl*” describing knowledge about famous cartoon characters (e.g., *Tom*, *Jerry*) and their relationships (e.g., *First_Appearance_in*, *Has_desire*). In the following sections, in order to help users easily understand how OntoXpl works, we are going to illustrate the application in six steps: main menu functions, natural language explanation for concepts, roles and instances, inference information, general axiom browsing, Abox querying, and taxonomy of classes and roles.

3.1. Main menu functions

The OntoXpl system is built by referencing implementation-compliant web tier components from Java Server Pages (JSP) and Java Servlet extensions based on the tomcat server environment. At server sides, users should install Tomcat 4.0 (or higher version) and JDK1.4 (or higher version) and setup all the necessary environmental parameters such as “JAVA_HOME” in their operating systems in order to run OntoXpl successfully (details about system architecture are provided in Chapter 5). The web-based client-server architecture provides a convenient way for users to browse information with standard HTML browsers installed on computers. In a way, it is not required for users to have java environment installed at client sides. In addition, users need to connect to the OWL reasoner – Racer [24] before starting OntoXpl since it needs to connect and communicate with Racer, compute on the fly by generating and sending a series of reasoning commands to Racer and analyzing reasoning results from the reasoner as users interact with the system, and display results to users through web interfaces. Details about how to setup the environment and examples about loading ontologies into the OntoXpl system is given in Appendix A.1. Figure 3.1 displays the beginning welcome page of OntoXpl.

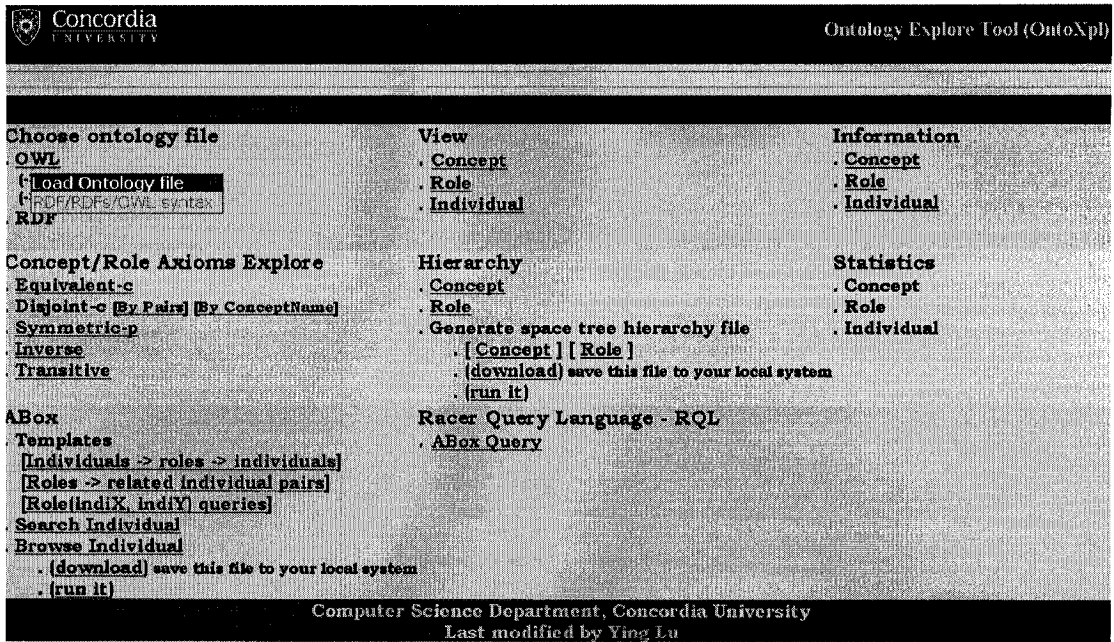


Figure 3.1: OntoXpl's main page

In order to let OntoXpl analyze and understand an ontology file, it is required that users load the ontology file to OntoXpl system first. Figure 3.2 shows the interface using which users can load ontologies.

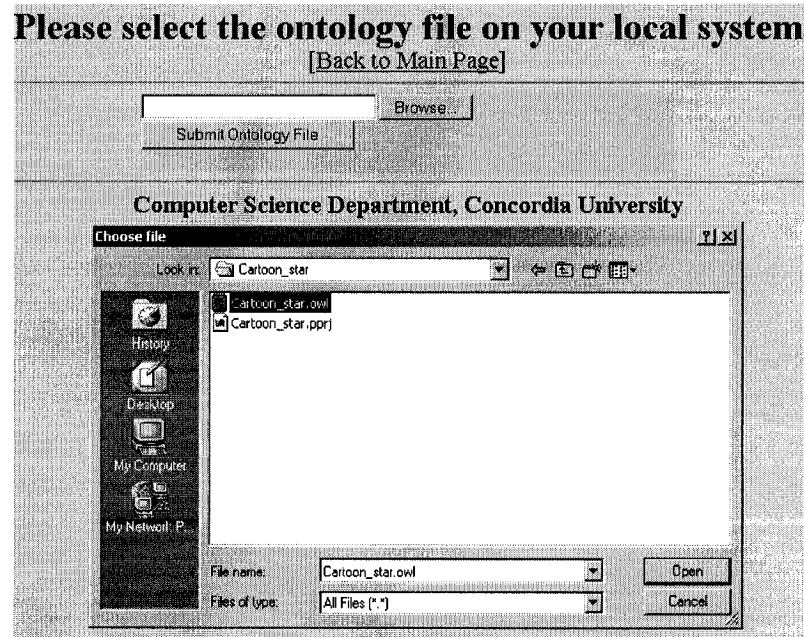


Figure 3.2: Ontology loading page

After an ontology has been loaded into the OntoXpl system successfully, the main command pane is shown in Figure 3.3. The filename of the OWL ontology currently loaded into OntoXpl and Racer is shown with a summary of the number of contained concept and role names (see also the following sections for an explanation of the example knowledge base). In our example ontology, the Tbox name is “*cartoon_star.owl*”, and there exist twenty-two *named-concepts*, sixteen *object properties*, and one *datatype property* as shown in Figure 3.3.

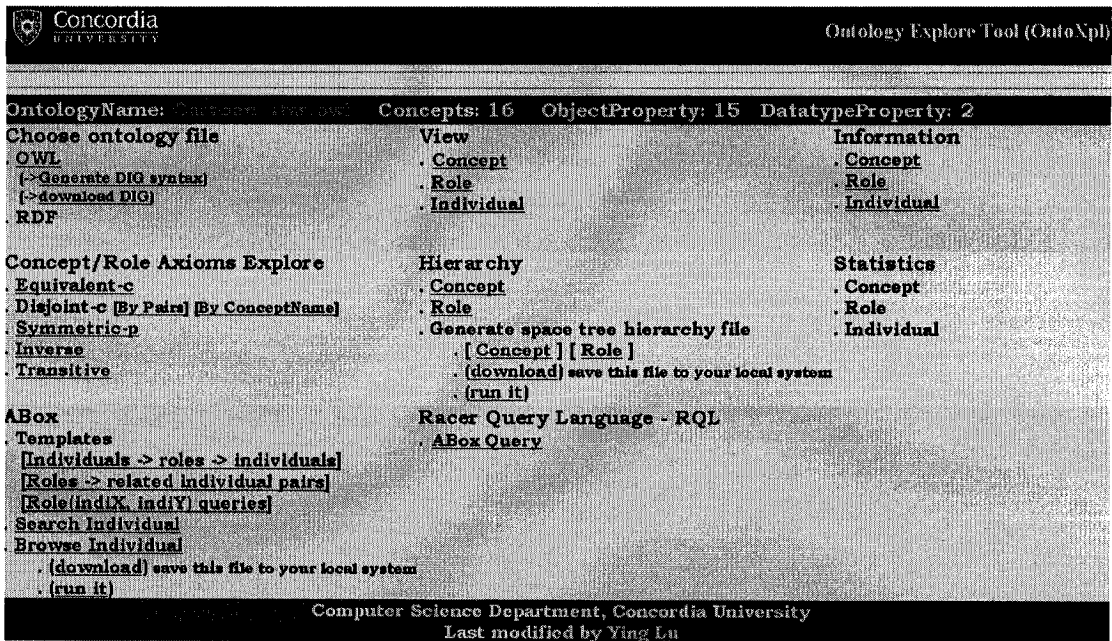


Figure 3.3: OntoXpl’s main page with the displayed ontology information

OntoXpl’s interface offers eight principal browsing categories (from left to right and top to bottom): file selector, “natural language” description, structural information, exploration of concept/property axioms, inspection of concept and role hierarchies, view of statistical information (not yet implemented), inspection of A-box graph structures, and the interactive use of Racer’s query language nRQL. In the following, the seven implemented categories are described. Figure 3.4 shows a zoom of the first (horizontal) command pane. The left group of commands is used to load an OWL file and generate a DIG representation of the loaded OWL file. The middle group of commands applies to concepts, roles, and individuals. Details and examples are displayed in the following chapters.

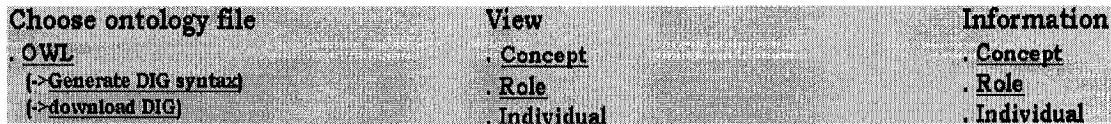


Figure 3.4: Zoom of the upper three command menus (from left to right): file selection, OWL / natural language views, information page views

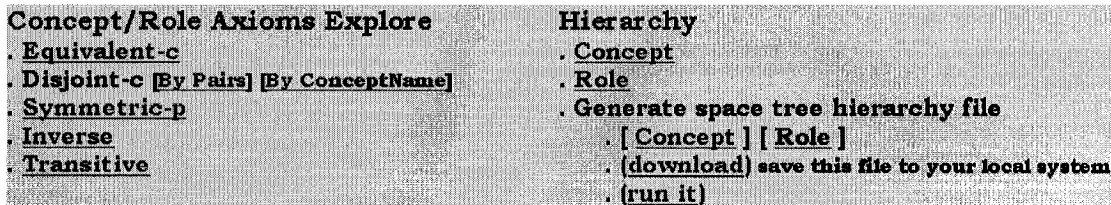


Figure 3.5: Zoom of the middle two command menus (from left to right): explore concept/property characteristics, show concept/role hierarchies

The two implemented command groups from the middle pane are shown in Figure 3.5. The command group displayed on the left allows one to query about equivalent or disjoint concept names and symmetric, inverse, and transitive roles. The other group is concerned with concept and role hierarchies. There exist two principal services: (i) one can browse the concept or roles hierarchies in an outline view; (ii) a data file for the Spacetree tool [13] is generated such that the taxonomies can be graphically inspected. The last two command groups from the bottom pane are shown in Figure 3.6. They are dedicated to explore A-boxes. The first command group has several search forms to retrieve individuals and their known relationships with other individuals, to browse relationships in an outline view or inspect the A-box structure with Spacetree. The second command group allows users to query A-boxes with Racer's query language nRQL [26].

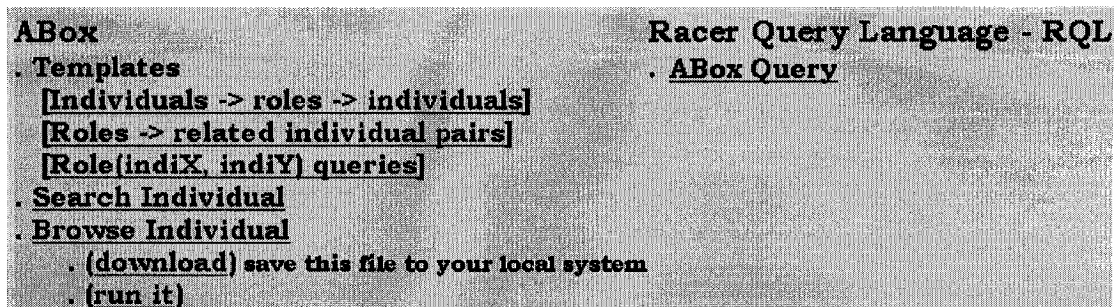


Figure 3.6: Zoom of the bottom two command menus (from left to right): A-box command menu, nRQL Racer Query Language

3.2. Natural language explanation

As the basic knowledge representation service, natural language explanation with less technical things provides a good way for more users, who do not have strong OWL background, to understand the object definition.

The OntoXpl system provides a user-friendly GUI. For example, when the mouse is over the menu item functional link, a dynamically generated layer containing explanation that is more informative will be shown to users. When the mouse moves away, the dynamic-generated layer will disappear. For example, when the mouse comes across the “Concept” hyperlink, the floating layer “Select one’s interest concept - with NL explanation” will be shown.

The view menu item shown in Figure 3.7 includes three parts: concept, role, and individual. We are going to explain them one by one. Under each function, we provide both the OWL definition and Natural Language (NL) information. In order to generate

the NL explanation, we parse and analyze the OWL syntax, format and display the desired information in a manner convenient to users so that users can easily understand how the selected concept, role and individual is defined.

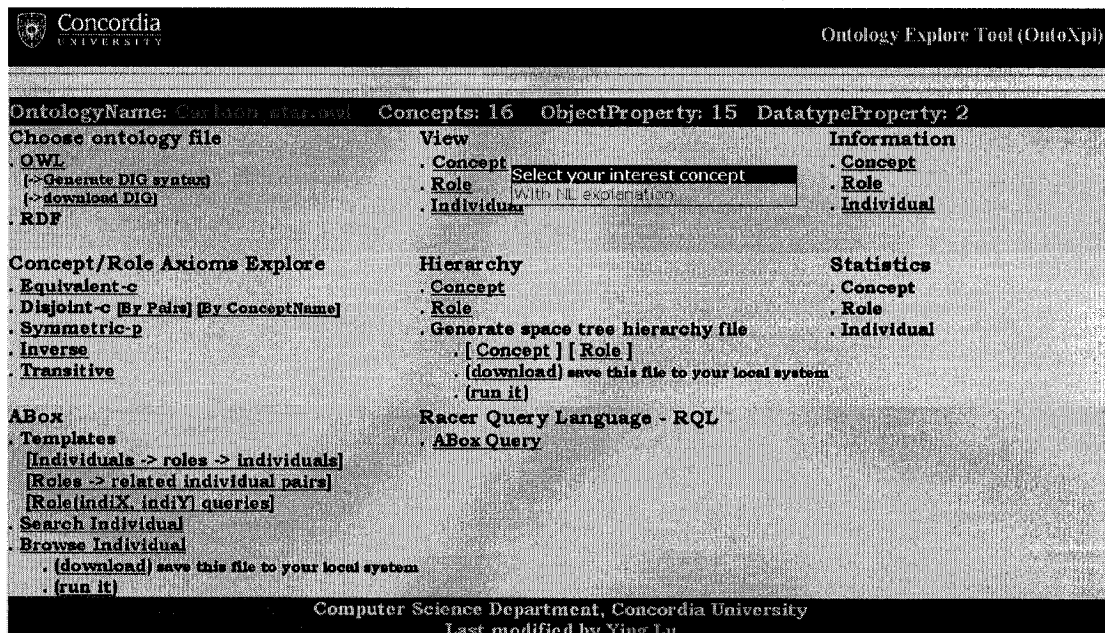


Figure 3.7: View function: Concepts, Roles and Individuals

Below the Concept-NL part, we list all concepts defined or used by the loaded ontology file. We alphabetically order the concept names so that it is easier for users to search for a given name. In the left frame (Since the OntoXpl system is developed based on the web, in this example, we separate the screen into two frames. Details are in Chapter 5 in the system architecture part), the top level locates the total number of concepts in the domain such as “Total concept number: 16” shown in Figure 3.8. Besides, in order to ease users’ reading and searching for the concepts, we divide the concepts into a small ten-record size concept groups with a boundary line separating them.

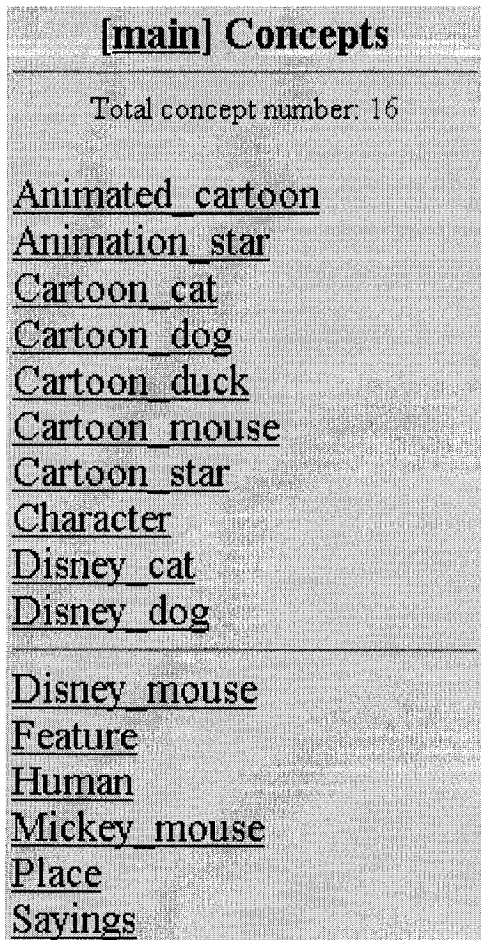


Figure 3.8: List Concepts in the domain

Because of the hyperlink support, web application has become very popular. As a result, it is easy to link objects together. Each concept name in Figure 3.8 is represented as a functional hyperlink, which means, whenever users click the link, the OWL source code (e.g., see Figure 3.9) together with a “natural language” description (e.g., see Figure 3.10) for the selected concept are shown on the right hand frame.

```
(owl:disjointWith rdf:resource="#Cartoon_mouse")
(/owl:disjointWith)
(owl:disjointWith rdf:resource="#Cartoon_dog")
(/owl:disjointWith)
(rdfs:subClassOf rdf:resource="#Cartoon_star")
(/rdfs:subClassOf)
```

Figure 3.9: OWL specification of class "*Cartoon_cat*"

```
this class has no individuals in common with Cartoon_mouse
this class has no individuals in common with Cartoon_dog
It is the subclass of Cartoon_star
```

Figure 3.10: "Natural Language" description of class "Cartoon_cat"

The "natural language" (NL) description is based on the DL notation and tries to describe the selected item w.r.t. this notation. These NL descriptions are intended for users with a limited background on DL and OWL.

Figure 3.11 shows the whole interface of the OWL source and natural language descriptions for class "*Cartoon_cat*". Within the page, the default namespace is displayed. Also, there is an "Info Page" link, which will show how the other concepts, roles, and instances are related to the current concept ("Concept Info" examples will be shown in Chapter 3.3). Below the NL part, every meaningful item defined in the domain such as concept, role and individual names are shown as functional hyperlinks through which users can navigate through the pages that are related to the selected item.

Current Concept is: <u>Cartoon_cat</u> [info Page]	
The namespace for this concept is: http://a.com/ontology	
owl definition	NL Explanation
<pre>(owl:disjointWith rdf:resource="#Cartoon_mouse") (/owl:disjointWith) (owl:disjointWith rdf:resource="#Cartoon_dog") (/owl:disjointWith) (rdfs:subClassOf rdf:resource="#Cartoon_star") (/rdfs:subClassOf)</pre>	<p>this class has no individuals in common with Cartoon_mouse</p> <p>this class has no individuals in common with Cartoon_dog</p> <p>It is the subclass of Cartoon_star</p>

Figure 3.11: “*Cartoon_cat*” OWL definition and NL explanation main page

After clicking “*Cartoon_mouse*”, we will go to the concept “*Cartoon_mouse*” NL description page shown in Figure 3.12. Actually, there is not much difference from Figure 3.11 in that they use the same functions and interface provided by OntoXpl, and only the displayed results are generated and computed on the fly. Within the NL descriptions, whenever concepts, roles, or instances are mentioned, we will provide a clickable link for them. For instance, the NL exploration for “*Cartoon_mouse*” shown in Figure 3.13 includes a named class “*Cartoon_dog*”, and the concept “*Cartoon_dog*” is a clickable link. After users click this link, the current selected concept will change to “*Cartoon_dog*”.

Current Concept is: <u>Cartoon_mouse</u> [info Page]	
The namespace for this concept is: http://fa.com/ontology	
owl definition	NL Explanation
<pre> (rdfs:subClassOf) (owl:Class rdf:about="#Cartoon_star") (/owl:Class) (/rdfs:subClassOf) (owl:disjointWith) (owl:Class rdf:about="#Cartoon_cat") (/owl:Class) (/owl:disjointWith) (owl:disjointWith) (owl:Class rdf:about="#Cartoon_dog") (/owl:Class) (/owl:disjointWith) (rdfs:subClassOf) (owl:Restriction) (owl:allValuesFrom) (owl:Class rdf:about="#Disney_mouse") (/owl:Class) (/owl:allValuesFrom) (owl:onProperty) (owl:SymmetricProperty rdf:about="#Is_cousin_of") (/owl:SymmetricProperty) (owl:onProperty) (owl:Restriction) (owl:allValuesFrom) (owl:Class rdf:about="#Disney_mouse") (/owl:Class) (/owl:allValuesFrom) (/owl:onProperty) (/owl:Restriction) (/rdfs:subClassOf) </pre>	<p>It is the anonymous subclass of A concept <u>Cartoon_star</u> this class has no individuals in common with : A concept <u>Cartoon_cat</u> this class has no individuals in common with : A concept <u>Cartoon_dog</u> It is the anonymous subclass of all of its instances have to be: A concept <u>Disney_mouse</u> for Symmetric property <u>Is_cousin_of</u></p>

Figure 3.12: “*Cartoon_mouse*” OWL definition and NL explanation

```

It is the anonymous subclass of
A concept Cartoon_star
this class has no individuals in common with :
A concept Cartoon_cat
this class has no individuals in common with :
A concept Cartoon_dog
It is the anonymous subclass of
  all of its instances have to be:
    A concept Disney_mouse
      for Symmetric property Is_cousin_of

```

Figure 3.13: “*Cartoon_mouse*” NL-Explanation

Cartoon_mouse \sqsubseteq Cartoon_star
Cartoon_mouse $\sqsubseteq \forall$ Is_cousin_of.Disney_mouse
Cartoon_mouse \sqcap Cartoon_cat \sqsubseteq Bottom
Cartoon_mouse \sqcap Cartoon_dog \sqsubseteq Bottom

Figure 3.14: “*Cartoon_mouse*” DL-Definition

About the Empirical facts that we mentioned in section 2.2, here we give an example, showing how it works in the system as shown in Figure 3.15.

Current Concept is: Feature [info Page]	
The namespace for this concept is: http://a.com/ontology	
owl definition	NL Explanation
This concept might be the Empirical Fact within the domain	

Figure 3.15: Empirical fact example “Feature”

Similarly to the concept-NL function, in Figures 3.16 and 3.18, we give two examples about how role-NL and individual-NL work. We select “*In_same_cartoon_series*” as a role example whose namespace is “<http://a.com/ontology>”. The NL definition is shown in Figure 3.17. Comparing with the OWL syntax, the Natural language is much easier to understand for people with restricted or no background of OWL and logic symbols.

Current Role is: In same cartoon series [info Page]	
The namespace for this role is: http://a.com/ontology	
owl definition	NL Explanation
<pre>(rdfs:domain rdfs:resource="#Cartoon_star") (/rdfs:domain) (rdfs:type rdfs:resource="http://www.w3.org/2002/07/owl#TransitiveProperty") (/rdfs:type) (rdfs:subPropertyOf) (owl:ObjectProperty rdfs:about="#In_same_series") (/owl:ObjectProperty) (/rdfs:subPropertyOf) (rdfs:range rdfs:resource="#Cartoon_star") (/rdfs:range) (rdfs:type rdfs:resource="http://www.w3.org/2002/07/owl#SymmetricProperty") (/rdfs:type)</pre>	<pre>Domain of this Role is: Cartoon_star Type of this Role is: http://www.w3.org/2002/07/owl TransitiveProperty Parent Property is: In_same_series Range of this Role is: Cartoon_star Type of this Role is: http://www.w3.org/2002/07/owl SymmetricProperty</pre>

Figure 3.16: “*In_same_cartoon_series*” Role NL-explanation main page

Domain of this Role is: Cartoon_star
 Type of this Role is: <http://www.w3.org/2002/07/owl> TransitiveProperty
 Parent Property is:
In_same_series
 Range of this Role is: Cartoon_star
 Type of this Role is: <http://www.w3.org/2002/07/owl> SymmetricProperty

Figure 3.17 NL-explanation for Role “*In_same_cartoon_series*”

Figure 3.18 shows the Individual NL-explanation main page. We select “*Mickey*” the famous Disney mouse as the example. Enlarged picture for the NL part is shown in Figure 3.19. Details about the OWL language parser, natural language translation rules, and the algorithms can be found in Chapter 5.

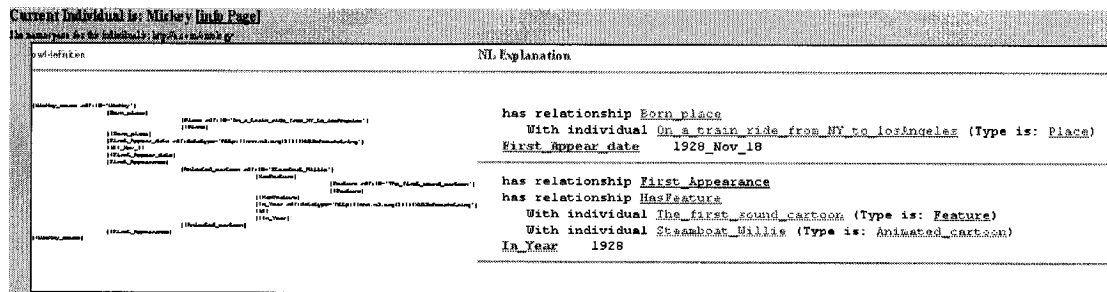


Figure 3.18: “*Mickey*” Individual NL-explanation main page

```

has relationship Born_place
  With individual On_a_train_ride_from_NY_to_losAngeles (Type is: Place)
First_Appear_date 1928_Nov_18
-----
has relationship First Appearance
has relationship HasFeature
  With individual The_first_sound_cartoon (Type is: Feature)
  With individual Steamboat_Willie (Type is: Animated_cartoon)
In_Year 1928
  
```

Figure 3.19: NL-explanation for Instance “*Mickey*”

3.3. Inferred information

The information views of concepts, roles, and individuals use Racer's query interface to display their (inferred) characteristics. Concepts are described by (i) their relative position in the classification hierarchy (e.g., parent, children), (ii) the roles occurring in the concept declarations, and (iii) the individuals that are instances of this concept. By analogy, a role is similarly described but in addition to its position in the role hierarchy, the concepts that use this role are listed. An individual is described by (i) its most specific concept names (so-called types) of which it is an instance, (ii) other individuals that are instances of concepts (parents, children, etc) related to its types.

3.3.1. Concepts level

Figure 3.19. displays the current concept name "*Cartoon_mouse*", its namespace "*http://a.com/ontology*", and the "NL Expl Page" link with contrast to the "Info page" under Concept-NL main page. Below that, inference results for the current selected concept are displayed.

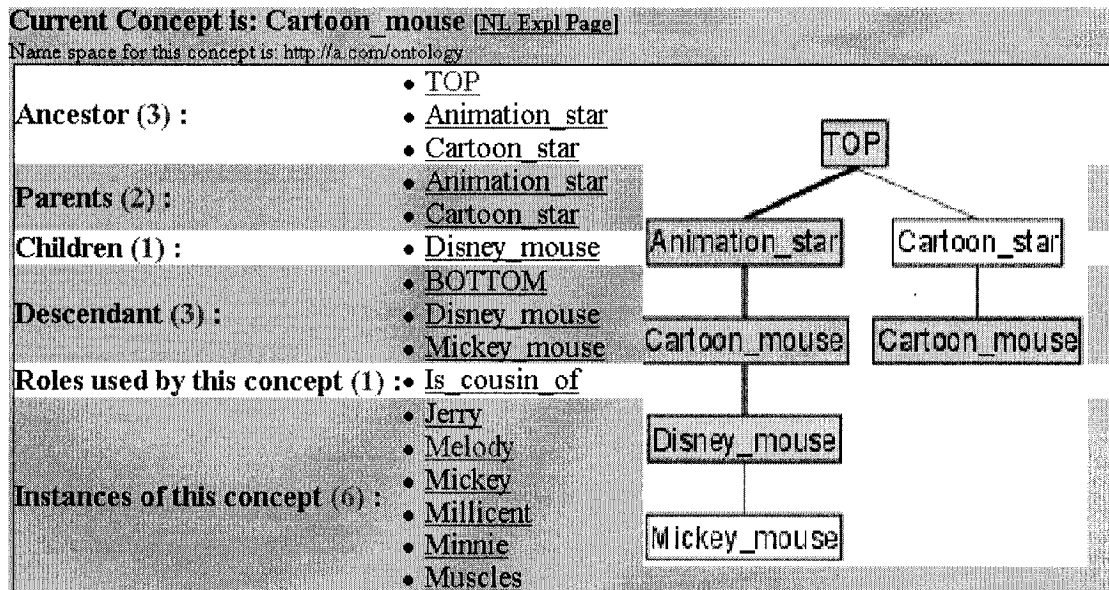


Figure 3.20: “*Cartoon_mouse*” Concept Information Page

As shown in Figure 3.20, we provide the names of ancestors, parents, children, descendants, roles, and individuals information about the current selected concept with staggered color background between them. In this example, we can see that “*Cartoon_mouse*” has two parents “*Animation_star*” and “*Cartoon_star*”, three ancestor concepts, one direct child, and three descendants. Each of them is displayed as a hyperlink. One can click the link and change the current selected concept. In addition, “*Cartoon_mouse*” uses one role: “*Is_cousin_of*” which is shown as a hyperlink as well. In contrast with a concept link, after we click a role link, it will go to the role inference information page with the clicked role as the current selected role. The final part lists all instances related to the selected concept. In the domain, we have a total of six individuals that are instances of “*Cartoon_mouse*”: individuals are “*Jerry*”, “*Melody*”, “*Mickey*”, “*Millicent*”, “*Minnie*”, and “*Muscles*”.

The “NL Expl page” link in the top of the right frame can take users back to the concept-NL main page. For example, after clicking “NL Expl Page” as shown in Figure 3.20, users will go back to the “*Cartoon_mouse*” Concept-NL page displayed in Figure 3.12.

3.3.2. Role level

In the role information inference part, users will see ancestors, parents, children and descendant roles for the current selected role. In addition, we provide a summary list of concepts that use the current selected role. In Figure 3.21, we can see that both the children level and the parent level roles’ results are NIL. However, NIL does not mean “it does not exist”, but “cannot be proven w.r.t. the information given to Racer”. Therefore, the ancestor level and the descendant level roles for “*Voiced_by*” are empty. In addition, there is only one concept “*Cartoon_animal*” which uses role “*Voiced_by*”.

[main] Role	Current Role is: Voiced_by [NL Expl Page]
Total object property: 15	Name space for this role is: http://a.com/ontology
Born place	Ancestor (0) :
Favorite sayings	Parents (0) : • NIL
First Appearance	Children (0) : • NIL
HasCharacter	Descendant (0) :
HasFeature	Concepts use this role (1) : • Cartoon star
Has Appetizer	Concordia University, Computer Science Department
Has nieces	
In same cartoon series	
In same series	
Is Appetizer for	
Is cousin of	
Is friend of	
Is niece of	
Not fond of	
Voiced by	
Total Datatype property: 2	
First Appear date	
In Year	

Figure 3.21: “Voiced_by” role information page

Another example about the role information page is “*In_same_series*”. After clicking the “*In_same_series*” link in Figure 3.21, the current selected role will be changed to “*In_same_series*”, and the contents will be recomputed and generated on the fly. As shown in Figure 3.22, we can see “*In_same_series*” has one children level role “*In_same_cartoon_series*”, which is the only descendant role as well.

Current Role is: In_same_series [NL Expl Page]
Name space for this role is: http://a.com/ontology
Ancestor (0) :
Parents (0) : • NIL
Children (1) : • In same cartoon series
Descendant (1) : • In same cartoon series
Concepts use this role (0) :

Figure 3.22: “*In_same_series*” Individual Information Page

3.3.3. Instance level

As a summary, we have selected a flexible way to develop our GUI according to our situation. Web links help to link objects together in a convenient way to ease users' browsing the information and benefit the contents' arrangements and displays within limited space.

The main individual information page is shown in Figure 3.23. Functions provided by this page are: types of the current selected individual, sibling level instances, ancestor level instances, parents level instances, children level instances and descendant level instances. In the following examples, we are going to show how each of these functions works.

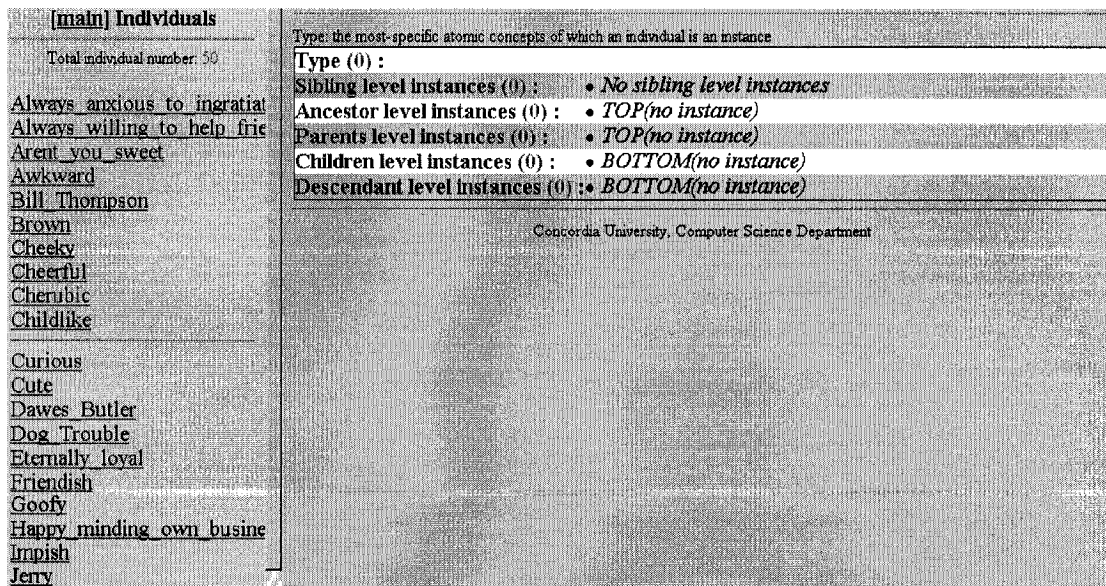


Figure 3.23: Individual Information main Page

In Figure 3.24, we select the individual “Minnie” as the first example. The type for instance “Minnie” is concept “Mickey_mouse”. The sibling level instances are “Melody”, “Mickey”, and “Millicent”. Since “Disney_mouse” is the parent concept for “Mickey_mouse”, and “Disney_mouse” has two instances “Jerry” and “Muscles”, the parent and ancestor level instances for “Minnie” are “Jerry” and “Muscles”.

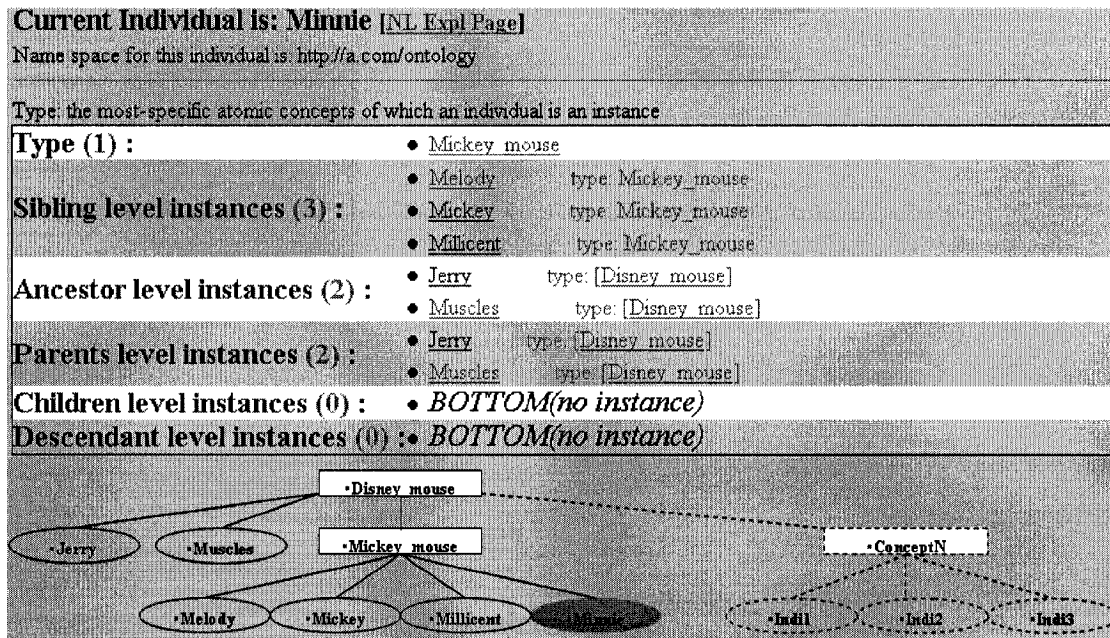


Figure 3.24: “Minnie” Individual Information page

Now we will use the following seven concepts: C1, C11, C12, C121, C13, C131, C1311, and their related instances to illustrate how we define type, sibling level instance, Ancestor level instance, Parent level instance, Children level instance, and Descendant level instances.

The concept hierarchy and instances information is shown in Figure 3.25: $C11 \sqsubseteq C1$, $C121 \sqsubseteq C12 \sqsubseteq C1$, $C1211 \sqsubseteq C121$, $C12121 \sqsubseteq C1212 \sqsubseteq C121$. C1 has instance “*Indi1*”, C11 has instance “*Indi11*”, C12 has “*Indi12*, *Indi13*”, C121 has “*Indi21*, *Indi22*, *Indi23*”, C1211 has “*Indi31*, *Indi32*”, C1212 has “*Indi33*”, and C12121 has instance “*Indi41*”. Here, we select “*Indi21*” as the example, whose most specific type is C121. According to the concept classification, we know the parent concept for C121 is C12, whose instances are “*Indi12*” and “*Indi13*”. Therefore, the parent level individuals are “*Indi12*” and “*Indi13*”. Similarly, we know the ancestor concepts for C121 are C1 and C12. As a result, individuals “*Indi1*”, “*Indi12*”, and “*Indi13*” are the ancestor level instances of “*Indi21*”. For the children level instances, we can see C1211 and C1212 are direct children concepts for C121. Therefore, we know that “*Indi31*”, “*Indi32*”, and “*Indi33*” are the results. In addition, when we migrate to the descendant part, we need to include the concept C12121 plus the children concepts mentioned above, and then we have “*Indi31*”, “*Indi32*”, “*Indi33*” and “*Indi41*” as the descendant level instances of “*Indi21*”. Figure 3.27 illustrates the OntoXpl instance information result for instance “*Indi21*”. This figure displayed here is not generated by OntoXpl, and we use it in order to illustrate what instance levels mean and how we present results for different level instances.

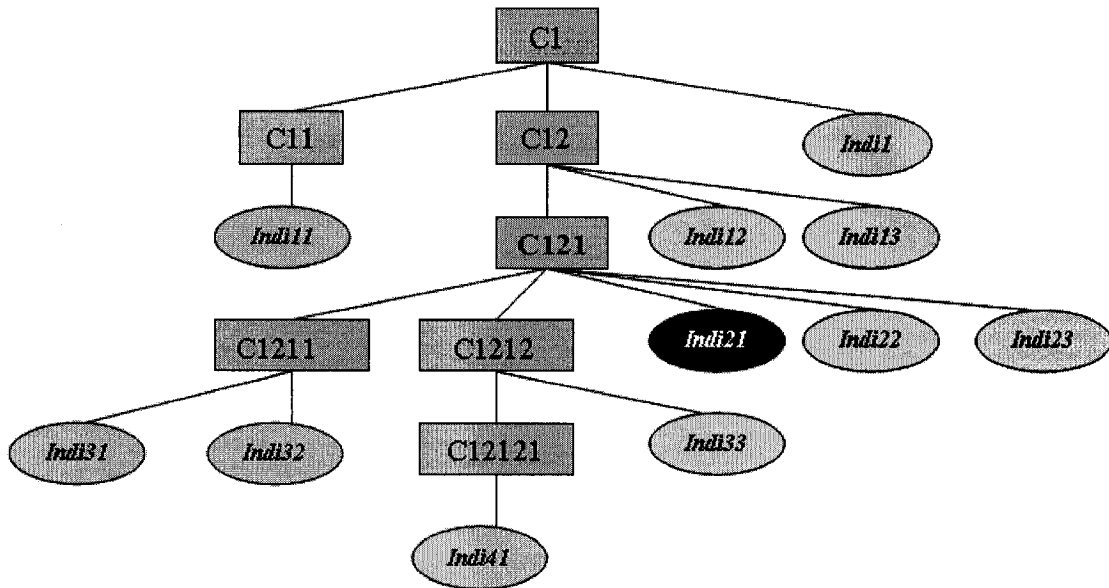


Figure 3.25: Case study example for “Individual Inference”

Current Instance	Type	Sibling level Instances	Ancestor level Instances	Parents level Instances	Children level Instances	Descendant level Instances
Indi21	C122	Indi22, Indi23	Indi1, Indi12, Indi13	Indi12, Indi13	Indi31, Indi32, Indi33	Indi31, Indi32, Indi33, Indi41

Figure 3.26: Case study result about Individual inference

[main] Individuals

Total individual number: 11

- [Indi1](#)
- [Indi11](#)
- [Indi12](#)
- [Indi13](#)
- [Indi21](#)
- [Indi22](#)
- [Indi23](#)
- [Indi31](#)
- [Indi32](#)
- [Indi33](#)
- [Indi41](#)

Current Individual is: Indi21 [NL Expl Page]

Name space for this individual is: <http://a.com/ontology>

Type: the most-specific atomic concepts of which an individual is an instance

Type (1) :	• C121	
Sibling level instances (2) :	• Indi22	type: C121
	• Indi23	type: C121
Ancestor level instances (3) :	• Indi1	type: [C1]
	• Indi12	type: [C12]
	• Indi13	type: [C12]
Parents level instances (2) :	• Indi12	type: [C12]
	• Indi13	type: [C12]
Children level instances (3) :	• Indi31	type: [C1211]
	• Indi32	type: [C1211]
	• Indi33	type: [C1212]
Descendant level instances (4) :	• Indi31	type: [C1211]
	• Indi32	type: [C1211]
	• Indi33	type: [C1212]
	• Indi41	type: [C12121]

Concordia University, Computer Science Department

Figure 3.27: OntoXpl instance-info result for “Indi21”

3.4. Axiom browsing

In this section, we provide the “Equivalent-concept”, “Disjoint-concept”, “Symmetric-role”, “Inverse-role” and “Transitive-role” functions. Each of these functions pre-calculates the related concepts or roles and then gives a summary report to users. For example, the “Equivalent-concept” function analyzes all concepts in the domain and gives back the equivalent classes result to users. If there are no equivalent concepts in the domain, an empty page will be seen. Now, we will explain these five functions one by one.

3.4.1. Equivalent concepts

In the `Cartoon_star` domain, there are two concepts: “*Animation_star*” and “*Cartoon_star*”, which are equivalent as shown in Figure 3.28.

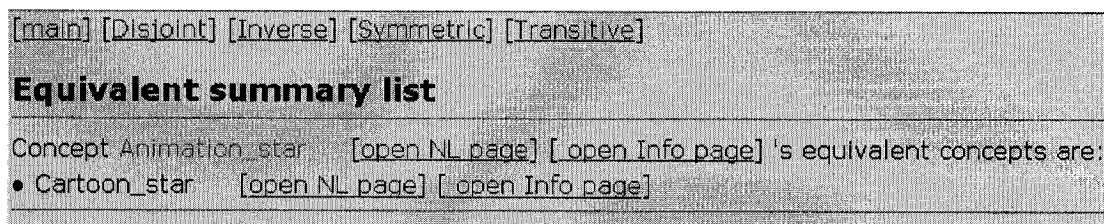


Figure 3.28: Equivalent-concept main page

3.4.2. Disjoint concepts

The concept disjoint function has been separated into two cases: “By conceptName” and “By pairs”. The first case is normally faster to compute. To be more precise, the complexity of this computation would be $O(n)$, where n is the number of concept names in the input ontology file. This complexity, however, is $O(n^2)$ for the second case, which we call “By pairs”. Figure 3.29 shows a part of the disjoint concept pairs, presented in ascending order based on the concept names. The second part gives feedback results according to users’ input. For example, in Figure 3.30, with keyword input “*Cartoon*”, all concepts whose name contains “*Cartoon*” will be listed as shown in Figure 3.31. Let us assume the concept of interest is “*Cartoon_cat*”. Disjoint information is that “*Cartoon_cat is disjoint with Cartoon_dog, Cartoon_mouse, Disney_dog, Disney_mouse, Human, and Mickey_mouse*” as shown in Figure 3.32. Every concept disjoint with “*Cartoon_cat*” is displayed as a link to help change the current selected disjoint concept. For instance, clicking “*Mickey_mouse*” in Figure 3.32, a new result screen is shown in Figure 3.33.

[main] [equivalent] [Inverse] [Symmetric] [Transitive]

Disjoint role pairs

- Animation_star=====Human
- Cartoon_cat=====Cartoon_dog
- Cartoon_cat=====Cartoon_mouse
- Cartoon_cat=====Disney_dog
- Cartoon_cat=====Disney_mouse
- Cartoon_cat=====Human
- Cartoon_cat=====Mickey_mouse
- Cartoon_dog=====Cartoon_cat
- Cartoon_dog=====Cartoon_mouse
- Cartoon_dog=====Disney_cat
- Cartoon_dog=====Disney_mouse
- Cartoon_dog=====Human
- Cartoon_dog=====Mickey_mouse
- Cartoon_duck=====Human
- Cartoon_mouse=====Cartoon_cat
- Cartoon_mouse=====Cartoon_dog
- Cartoon_mouse=====Disney_cat
- Cartoon_mouse=====Disney_dog
- Cartoon_mouse=====Human
- Cartoon_star=====Human
- Disney_cat=====Cartoon_dog
- Disney_cat=====Cartoon_mouse
- Disney_cat=====Disney_dog
- Disney_cat=====Disney_mouse
- Disney_cat=====Human
- Disney_cat=====Mickey_mouse
- Disney_dog=====Cartoon_cat
- Disney_dog=====Cartoon_mouse
- Disney_dog=====Disney_cat
- Disney_dog=====Disney_mouse

Figure 3.29: Disjoint concept by pair

[main] [equivalent] [Inverse] [Symmetric] [Transitive]

Concept Name:

Figure 3.30: Disjoint concept name input

[main] [equivalent] [Inverse] [Symmetric] [Transitive]

Concept Name:

Cartoon_cat

Cartoon_dog

Cartoon_duck

Cartoon_mouse

Cartoon_star

Figure 3.31: Search Concept with one word "Cartoon"

[main] [equivalent] [Inverse] [Symmetric] [Transitive]

Concept Name:

Concept '*Cartoon_cat*' disjoint concepts are:

Cartoon_dog

Cartoon_mouse

Disney_dog

Disney_mouse

Human

Hickey_mouse

Figure 3.32: Disjoint results - "*Cartoon_cat*"

[main] [equivalent] [Inverse] [Symmetric] [Transitive]

Concept Name:

Concept '**Mickey_mouse**' disjoint concepts are:

Figure 3.33: Disjoint result for “Mickey_mouse”

3.4.3. Symmetric roles

All symmetric roles defined in the domain will be listed in a summary report to users as shown in Figure 3.34. Beside the role name, there are two links. After clicking the first one, a new window about the role natural language page will be opened with the current role as the selected role. The other one opens the role information page in a new window.

[main] [equivalent] [Disjoint] [Inverse] [Transitive]
Symmetric Properties summary list
<ul style="list-style-type: none"> • In_same_cartoon_series [open NL page] [open Info page] • In_same_series [open NL page] [open Info page] • Is_cousin_of [open NL page] [open Info page] • Is_friend_of [open NL page] [open Info page]

Figure 3.34: Symmetric roles in a summary report

3.4.4. Inverse roles


OntoXpl provides functions to get all the inverse roles pair by pair as shown in Figure 3.35. Similar to the symmetric roles function, there are two clickable links as well. Function “open NL page” will open the pages “Role Natural Language” and “Role Information”.

[main] [equivalent] [Disjoint] [Symmetric] [Transitive]
Inverse Properties summary list
Role Has_Appetizer [open NL page] [open Info page] 's inverse role(s) is/are: <ul style="list-style-type: none"> • Is_Appetizer_for [open NL page] [open Info page]
Role Has_nieces [open NL page] [open Info page] 's Inverse role(s) is/are: <ul style="list-style-type: none"> • Is_niece_of [open NL page] [open Info page]

Figure 3.35: Inverse roles in a summary report

3.4.5. Transitive roles

All transitive roles defined in the domain will be shown in this part. For instance, Cartoon_animal ontology has three transitive roles: “*In_same_cartoon_series*”, “*In_same_series*”, and “*Is_cousin_of*” as shown in Figure 3.36.



The screenshot shows a summary report for transitive roles. At the top, there are navigation links: [main], [equivalent], [Disjoint], [Inverse], and [Symmetric]. Below these is the title "Transitive Roles summary list". A horizontal line separates the title from the list of roles. The list contains three items, each with a bullet point and two links: [open NL page] and [open Info page].

- *In_same_cartoon_series* [open NL page] [open Info page]
- *In_same_series* [open NL page] [open Info page]
- *Is_cousin_of* [open NL page] [open Info page]

Figure 3.36: Transitive role in a summary report

3.5. Abox queries

The Abox query has been separated into three parts. Firstly, three kinds of templates will be introduced about query individuals in the domain. The next is about how to query instances based on the instance name given by users. Finally, all instances and their relationships will be calculated and shown to users. Now these three functions will be introduced one by one.

3.5.1. Abox template I - “Individuals -> roles -> individuals”

The first template begins from all instances that have relationship with others. The goal is to help users catch relationships between instances. In a way, instances that are not linked or used by other instances have been excluded, and they are not displayed to users. Clicking an instance name, the system will calculate and generate roles used by the selected instance and display results in a list menu. In our example, as shown in Figure 3.37, there are thirteen instances that are related to other instances. After selecting “Goofy”, roles used by “Goofy” will be generated in a list menu.

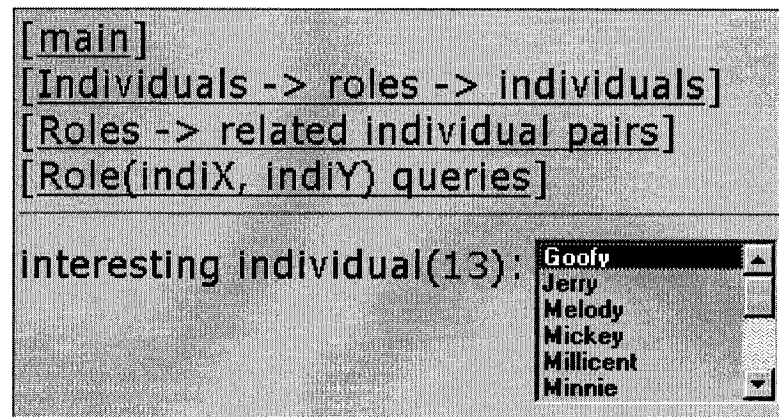


Figure 3.37: “Individuals->roles->individuals”: all instances related to other instances

After selecting “*In_same_cartoon_series*” role in the second list box, all the instances that are “*In_same_cartoon_series*” with “Goofy” are shown in Figure 3.38. “*Cartoon_star*” ontology defines “*In_same_cartoon_series*(Goofy, Mickey), *In_same_cartoon_series*(Mickey, Minnie), *In_same_cartoon_series*(Minnie, Millicent), *In_same_cartoon_series*(Minnie, Melody), and *In_same_cartoon_series*(Pluto, Goofy)” in Figure 3.38. Besides, “*In_same_cartoon_series*” is defined as not only a symmetric but a

transitive role. Therefore, the results for “Goofy” are “Melody”, “Mickey”, “Millicent”, “Minnie” and “Pluto” shown in Figure 3.39.

Direct Instances ▼ Goofy Pluto Spike	In Same Cartoon Series Mickey	Direct Instances ▼ Melody Mickey Millicent Minnie	In Same Cartoon Series Minnie
Direct Instances ▼ Melody Mickey Millicent Minnie	In Same Cartoon Series Millicent Melody	Direct Instances ▼ Goofy Pluto Spike	In Same Series Goofy

Figure 3.38: In_same_cartoon_series: (Goofy, Mickey)(Mickey, Minnie)(Minnie, Millicent)(Minnie, Melody)(Pluto, Goofy)

[main]
 [Individuals -> roles -> Individuals]
 [Roles -> related individual pairs]
 [Role(indiX, indiY) queries]

interesting individual(13):

individual 'http://a.com/ontology#Goofy' was selected

role 'http://a.com/ontology#In_same_cartoon_series' was selected

Goofy
 Jerry
 Melody
 Mickey
 Millicent
 Minnie

First Appearance
 HasCharacter
In same cartoon series
 In same series

Melody
 Mickey
 Millicent
 Minnie
 Pluto

Figure 3.39: Roles used by and instances related to “Goofy”

3.5.2. Abox template II - “Roles -> related individual pairs”

In template2, all the roles that are used by at least one instance will be shown in a list box first shown in Figure 3.40.

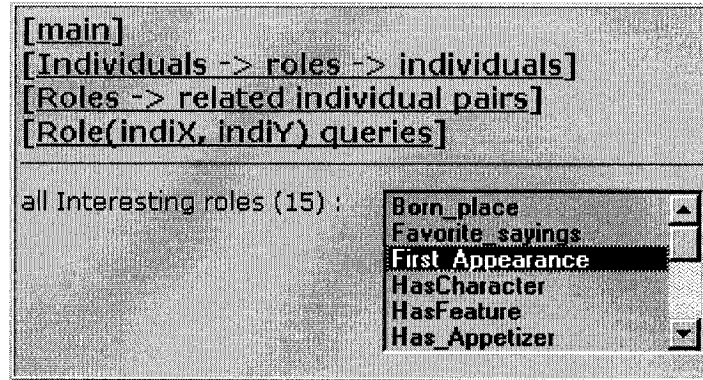


Figure 3.40: “Roles -> related individual pairs”: all roles used by instances

One of the interesting features that OntoXpl provides is to generate values based on a particular property. For example, suppose the user is interested about the property “*First Appearance*”. After a series of parsing and computing tasks, the system generates the results and provides a summary report, as shown in Figure 3.41.

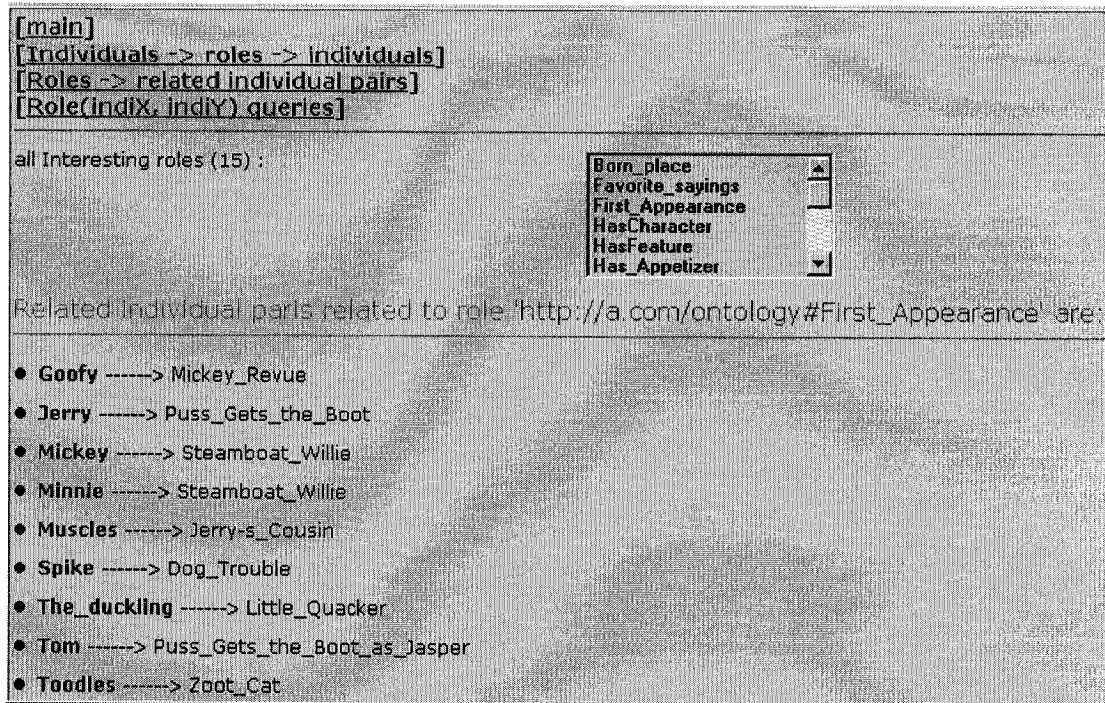


Figure 3.41: “*First Appearance*” Template II results

3.5.3. Abox template III - “Role(indiX, indiY) queries”

Template III helps answering “which roles have been used by instances at least N times and at most M times”. For instance, for roles used only once by instances, users can input at least 1, and at most 1 in the text field in Figure 4.32. The search results are “*Not_fond_of*”, “*HasFeature*”, and “*Born_place*”, as shown in Figure 3.43.

[main]
 [Individuals -> roles -> individuals]
 [Roles -> related individual pairs]
 [Role(indiX, indiY) queries]

role used by individuals: at least: time(s)
 at most: times(s)

Figure 3.42: Role(indiX, indiY) is used only once

[main]
 [Individuals -> roles -> individuals]
 [Roles -> related individual pairs]
 [Role(indiX, indiY) queries]

role used by individuals: at least: time(s)
 at most: times(s)

Not_fond_of [[open NL page](#)] [[open Info page](#)]
 • Spike-->Tom

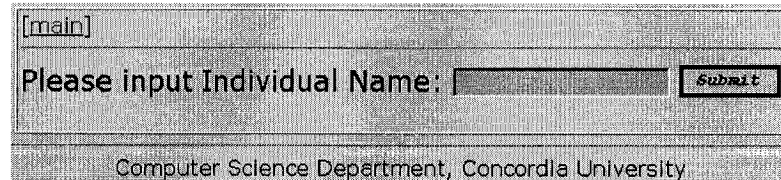
HasFeature [[open NL page](#)] [[open Info page](#)]
 • Steamboat_Willie-->The_first_sound_cartoon

Born_place [[open NL page](#)] [[open Info page](#)]
 • Mickey-->On_a_train_ride_from_NY_to_LosAngeles

Figure 3.43: Roles used only once by instances in the domain

3.5.4. Search by individuals

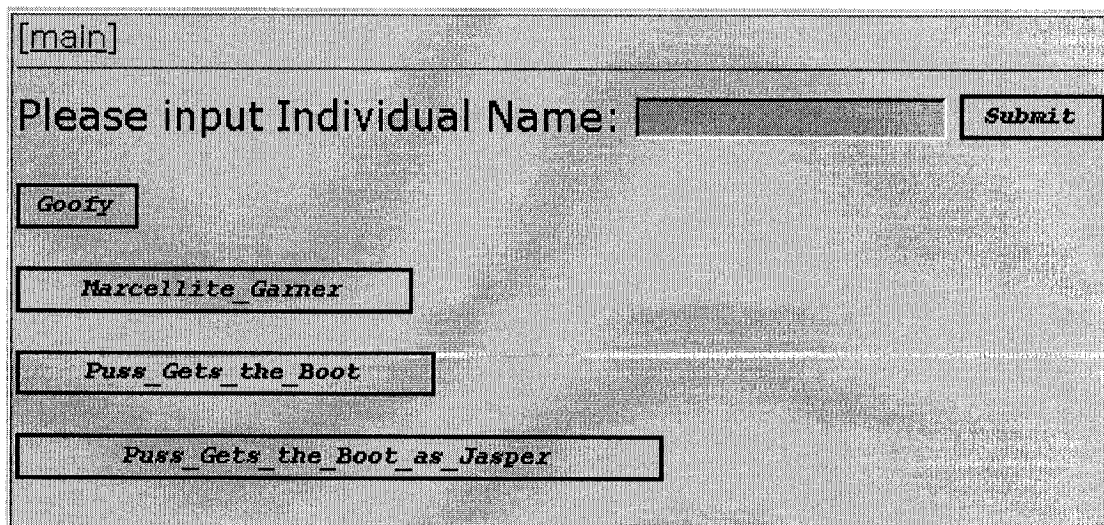
As in Figure 3.44, the individual name is the input in the text field. After clicking the submit button, individuals whose names contain users' input keywords will be listed.



The screenshot shows a web browser window with a title bar containing "[main]". Below the title bar, the text "Please input Individual Name:" is followed by a text input field and a "Submit" button. At the bottom of the browser window, the text "Computer Science Department, Concordia University" is visible.

Figure 3.44: Input Individual Name

The input value is G, which means the result values should return all instances whose name includes keyword 'G'. In the example, there are four instances, which satisfy the requirement. They are "Goofy", "Marcellite_Garner", "Puss_Gets_the_Boot" and "Puss_Gets_the_Boot_as_Jasper" as shown in Figure 3.45.



The screenshot shows the same web browser window as in Figure 3.44. The text input field now contains the letter "G". Below the input field, four individual names are listed, each in its own box: "Goofy", "Marcellite_Garner", "Puss_Gets_the_Boot", and "Puss_Gets_the_Boot_as_Jasper".

Figure 3.45: Individual name search results

After clicking “Goofy”, all information related to “Goofy” will be listed in two parts. One is ordered by the individual name; the other is ordered by role name. In the example, the result ordered by individual name is shown in Figure 3.46.

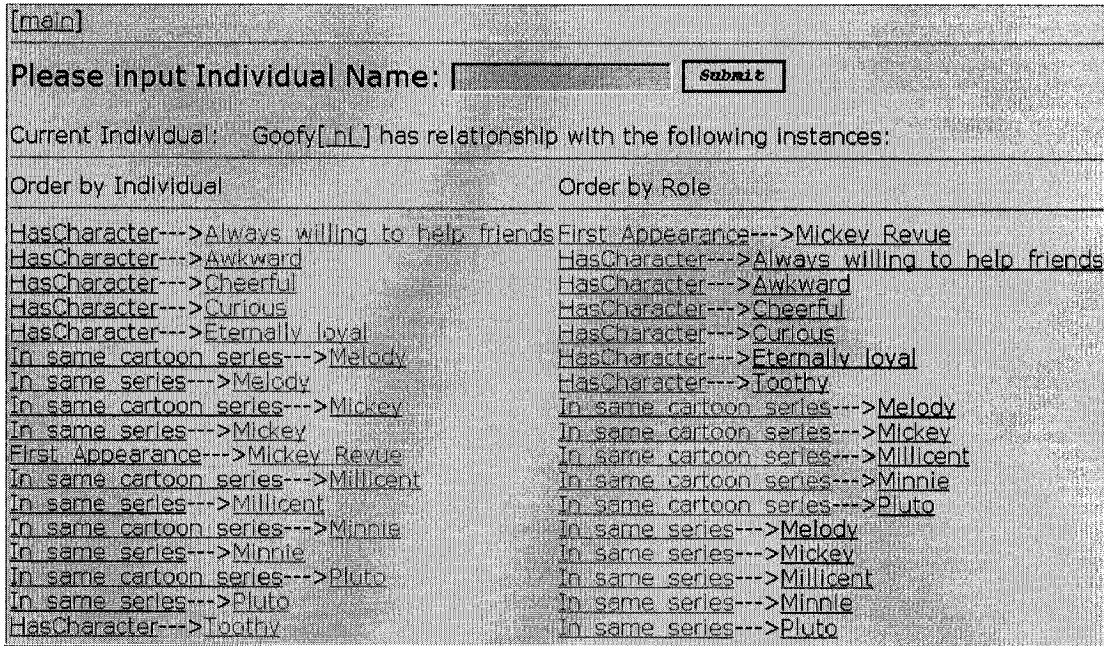


Figure 3.46: Search by Individual Name Result

When instances use datatype properties, the datatype information will be shown below the object property part. For instance, the individual “*The_duckling*” first appears in year 1950 – `First_Appear_date(The_duckling, #String(“1950”))` as shown in Figure 3.47.

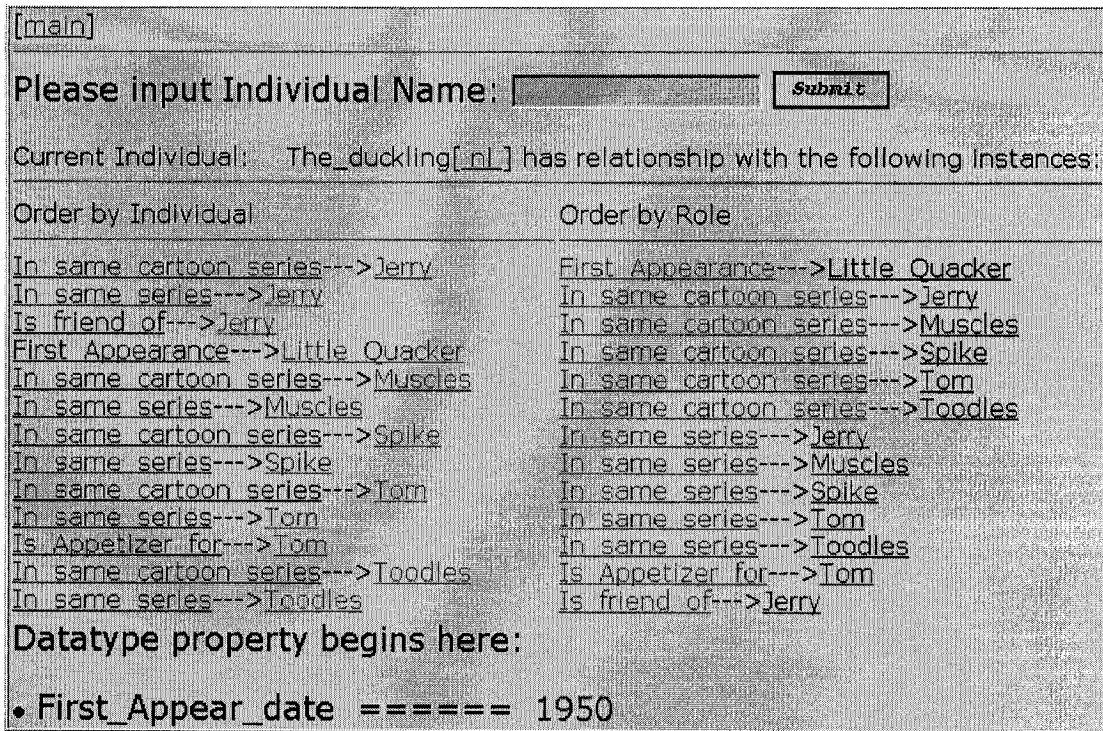


Figure 3.47: Individual using datatype role search result

3.5.5. Browse all instances

OntoXpl lists all instances that are related to others in red color, while the instances having no relationship with others are displayed in black. As shown in Figure 3.48, “Melody”, “Mickey”, “Millicent”, “Minnie”, and “Muscles” are individuals having relationship with others, so they are interesting instances for users. On the other hand, instances “Jerry-s_Cousin”, “Little_Quacker”, “Look_out_for_other_little_guy”, “Marcellite_Garner”, “Mickey_Revue” and “Oh_Mickey...” are possibly not interesting for users. Users can explicitly select the instance, OntoXpl will communicate with Racer and get back the roles, and instances used by the selected instance. In order to allow

users the management of the information efficiently, OntoXpl transforms the result into the space tree format (details about OntoXpl's outputs are presented in Chapter 4.4). Figure 3.49 shows the complete unfolded hierarchy for the individual example “Mickey” in an outline view.

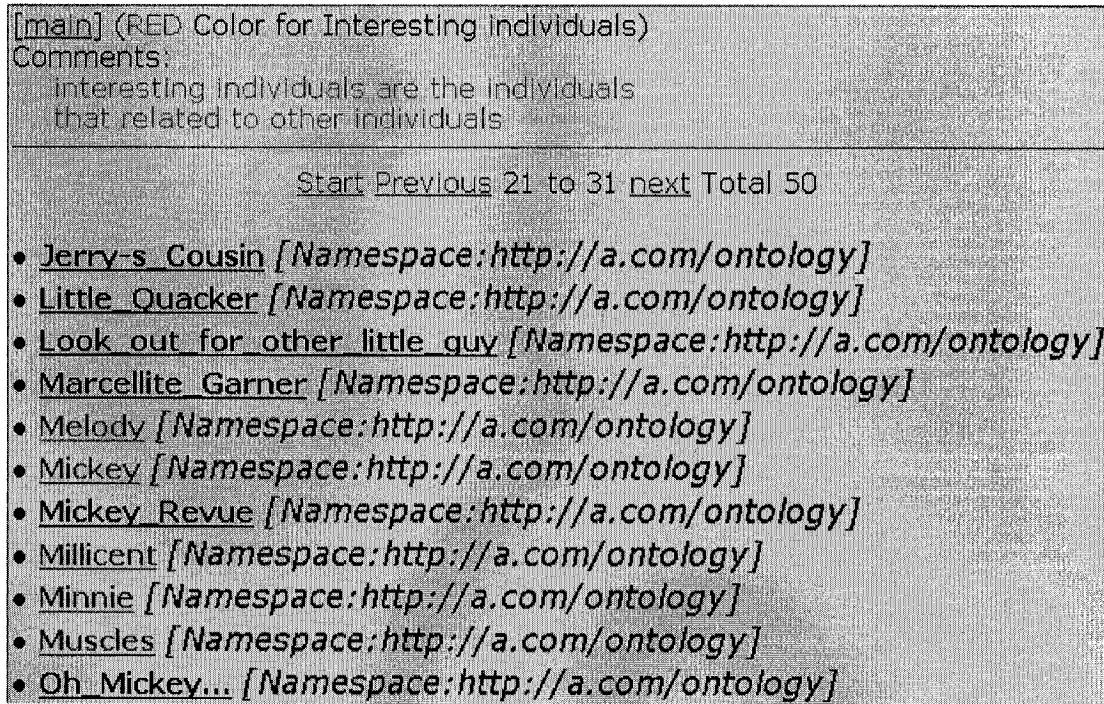


Figure 3.48: Browsing all instances

The disadvantage of this type of view is the repeated occurrence of classes (or subtrees) that have more than one parent (e.g., Goofy, Minnie). The current version of the Spacetree tool is version 1.6. Because of the limitation of this version, it does not support graph outputs. For example, there exists *In_same_cartoon_series(Mickey, Goofy)* and *In_same_series(Mickey, Goofy)*. Instead of putting and displaying them in a graph, the version 1.6 Spacetree's schema only supports displaying the results in a tree structure.

The improved algorithm is developing under the department of artificial intelligence at University of Ulm.

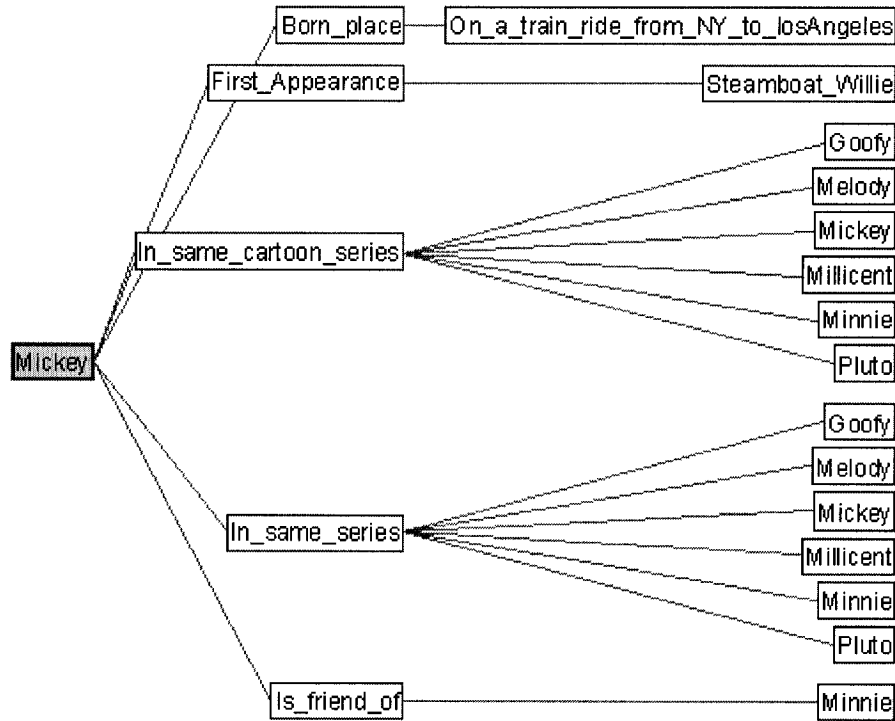


Figure 3.49: Space tree format output for instance “Mickey”

3.6. Taxonomy

Based on the concept’ definitions in the domain, the concept hierarchy result will be calculated and regenerated by OntoXpl system. Therefore, the final inference results can be different from users’ original design. Let us look at an example. C1, C2, C3, C31, and OR1 are defined as shown in Figure 3.50. The necessary and sufficient condition for C1 is “it has a filler in the role OR1, all of its instances have to be C3”, and the definition for

C2 is “for role *OR1*, all of its instances have to be A concept *C31*”. Thus, concept C2 should be the subclass of concept C1. The OntoXpl results are shown in Figure 3.51.

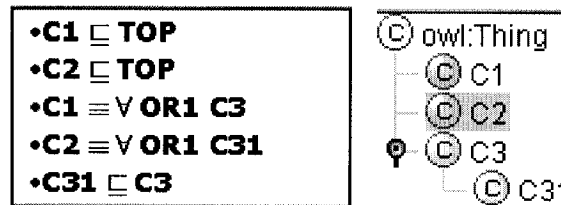


Figure 3.50: User’s concept definition

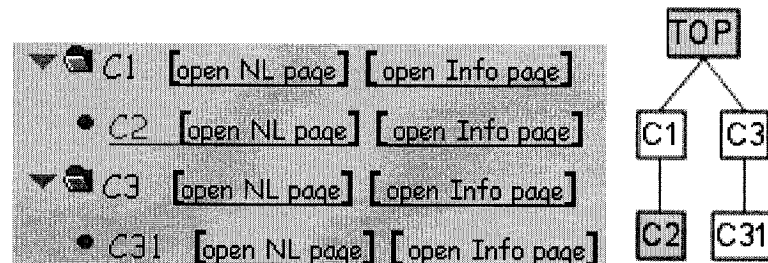


Figure 3.51: Concept hierarchy inference results

In contrast to the hierarchical views displayed by using the Spacetree tool shown in Figures 3.52 and 3.53, we provide another navigation mechanism for users browse concepts and roles’ hierarchy information based on JTree structure. Figures 3.54 and 3.55 show the complete unfolded hierarchy for concepts and roles in the *Cartoon_star* ontology using an outline view provided by the OntoXpl system. In case users need the NL description or more related information about the concept, we provide ways for users to go through those functions. In a way, our system develops an integrated display mechanism by providing and linking functions closely to help users achieve their goals.

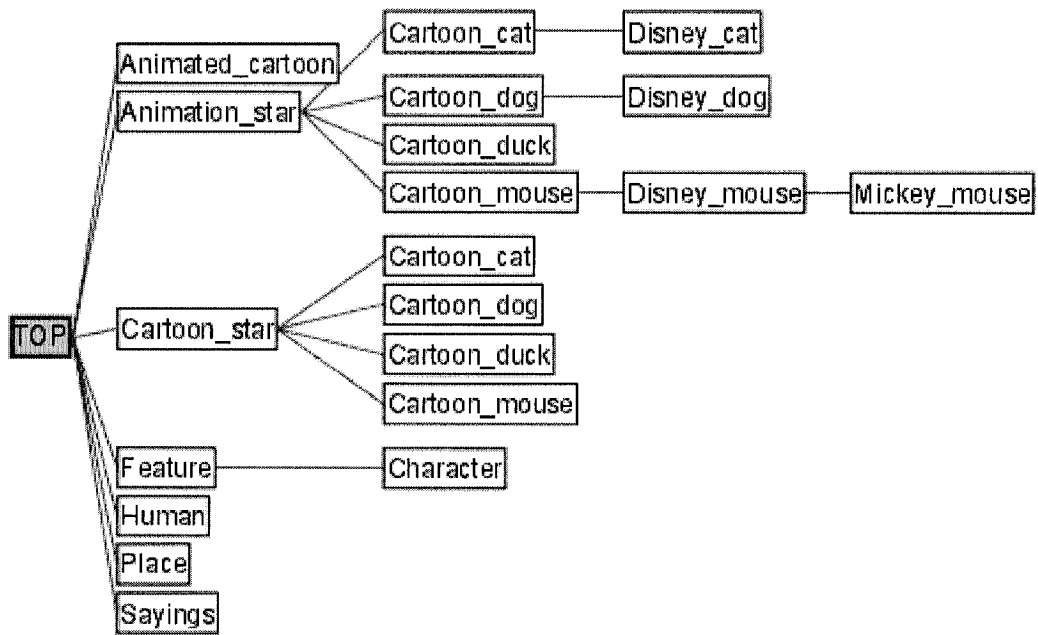


Figure 3.52: Spacetree concept hierarchy results

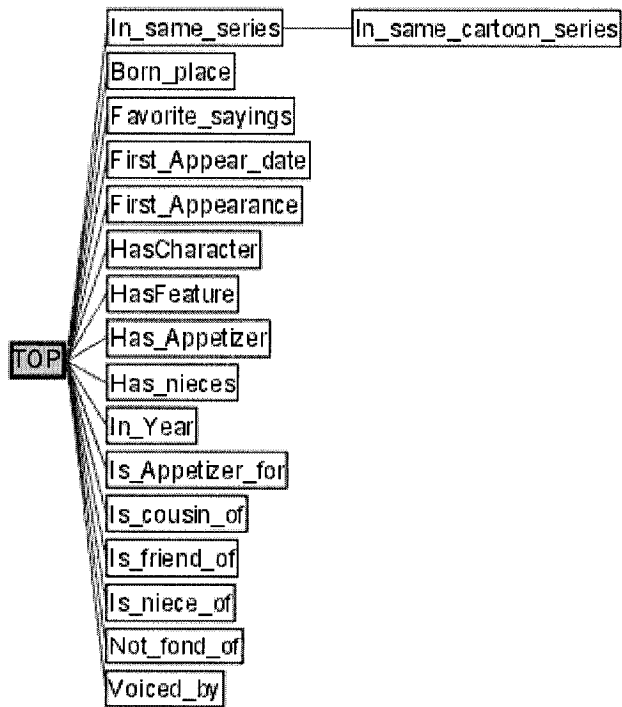


Figure 3.53: Spacetree role hierarchy results

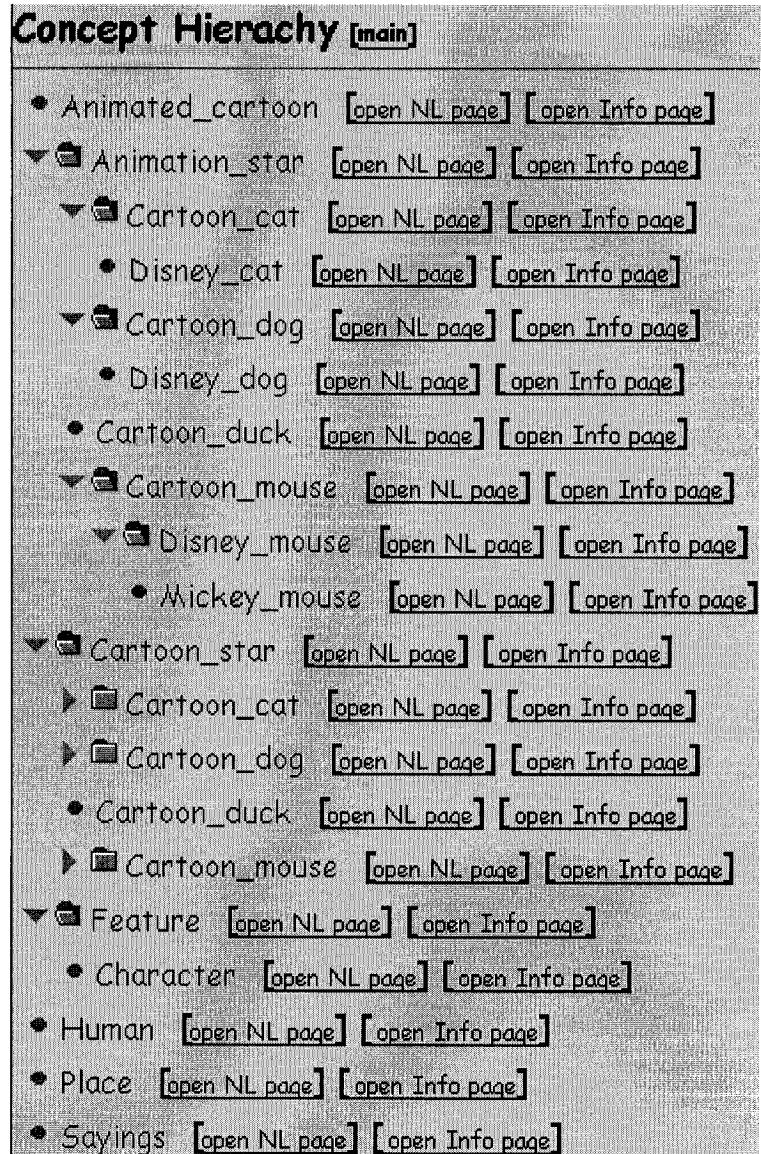


Figure 3.54: HTML/script Concept and role hierarchy



Figure 3.55: HTML/script Concept and role hierarchy

3.7. New Racer Query Language nRQL query interface

As an extended query language for Racer, nRQL [26] can be seen as a straightforward extension and combination of the Abox querying mechanisms to run the basic Abox retrieval function and allow the use of variables within queries, as well as much more complex queries formatted in a lisp-like notation. The variables in the queries are to be bound against those Abox individuals that satisfy the specified query. For expert users,

the OntoXpl system supports formulating queries in a formal language supported by Racer. We will guide users through the following two example scenarios in order to go through the nRQL part under OntoXpl.

The first scenario shown in Figure 3.56 describes the simple Abox query. In this example, users look for all instances that have been defined in the domain. The returned results are reorganized by hiding the namespace from the interface, and adding a functional link to help users retrieve more information related to instance names.

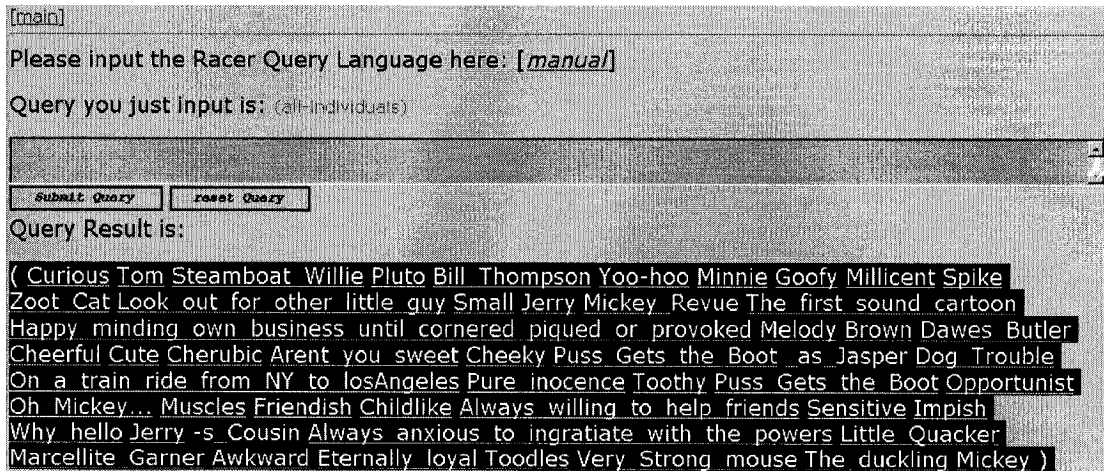


Figure 3.56: simple Abox query through nRQL interface

The second scenario shows a complex nRQL [26] query which searches for a “*Disney mouse*”, who has nieces, and is a friend of Disney mouse “*Mickey*”. The dialog box shown in Figure 3.58 displays the input query and its returned result in a Lisp-like notation. Figure 3.57 shows the DL definition of the query. There are two variables X and Y, whose types are “*Mickey_mouse*”. X is a niece of Y, and X is a friend of “*Mickey*”.

?X	: Mickey_mouse
?Y	: Mickey_mouse
{?X, Mickey}	: Is_friend_of
{?Y, ?X}	: Is_niece_of

Figure 3.57: Query “*Disney mouse*”, who has nieces, and is a friend of Mickey

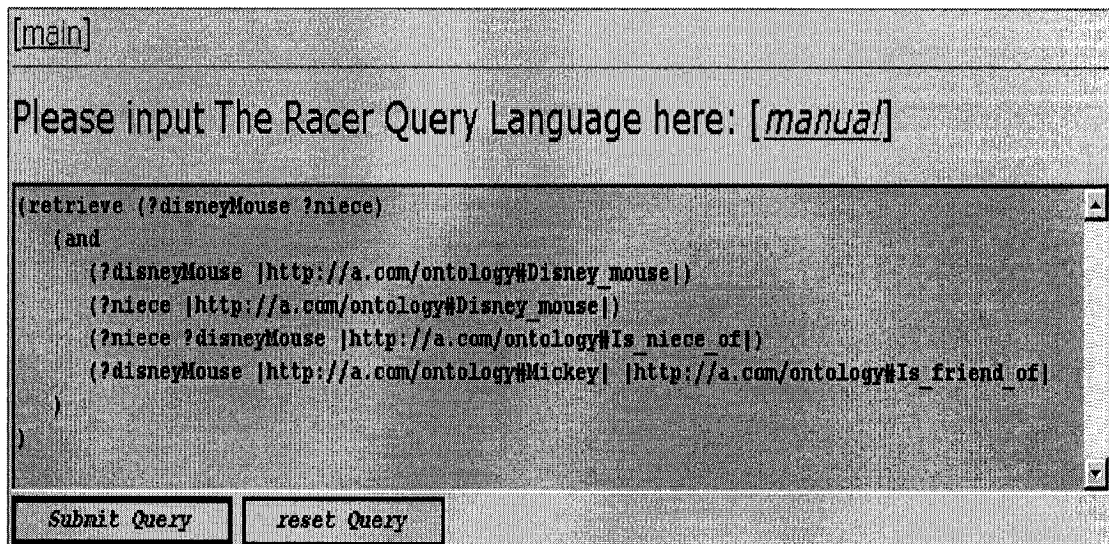


Figure 3.58: Example nRQL query and its result

After connecting and sending this requirement through Java API to Racer, the result is: “*Minnie*”, who is a friend of “*Mickey*”, has two nieces “*Millicent*” & “*Melody*” shown in Figure 3.59. In this example, the value for variable X is “*Minnie*”, and Y has a result set, whose values are {*Melody*, *Millicent*}.

Minnie	: Mickey_mouse
Millicent	: Mickey_mouse
Melody	: Mickey_mouse
<Minnie, Mickey>	: Is_friend_of
< Millicent, Minnie>	: Is_niece_of
< Melody, Minnie>	: Is_niece_of

Figure 3.59: Query results for scenario two

Figure 3.60 shows the results in a lisp-like notation through the OntoXpl interface. The variable “*disneyMouse*” equals “*Minnie*”, and variable “*niece*” holds two values “*Millicent*” and “*Melody*”. In addition, the namespaces have been hidden by OntoXpl but have been added to links, which will help users to get more detailed information.

```
Query Result is:
(((?DISNEYMOUSE Minnie) (?NIECE Millicent)) ((?DISNEYMOUSE Minnie) (?NIECE Melody)))
```

Figure 3.60: OntoXpl Abox complex query results in a lisp-like notation

4. Related work

Currently there are not many stable and usable ontology visualization or exploration tools (and even editors). The lack of suitable tools and their shortcomings were one of the major motivations for the design and implement OntoXpl. The motivation for OntoXpl's web server based architecture was ease of use with standard HTML browsers and simple adaptation to multi-user environments. To the best of our knowledge, OntoXpl is currently the only ontology exploration tool that is fully targeted to OWL and relies on Racer's deductive capabilities for offering users better exploration capabilities. OntoXpl system helps users to narrow down and retrieve the relevant information; we organize and display the desired information in a manner that is convenient to users. We retrieve and compute this information on the fly as the user interacts with the system.

As we have mentioned in Chapter 2, OntoXpl is designed as an ontology information-browsing tool instead of an ontology editor. OntoXpl is an interactive tool for organizing and browsing the knowledge in the domain. It helps users to browse and search the knowledge base. It also provides extensive inference functions to ensure the maximum exploration of the facts. OntoXpl can work well with OWL ontology editors such as Protégé and OilEd throughout the ontology development process. During the ontology development, ontology developers can quickly obtain a global view of the ontology file that they have implemented by loading the ontology file into OntoXpl and retrieving the information in the domain. This loading is done by typing the location of the ontology

file in the url field in the OntoXpl browser. In the following sections, the capabilities of Protégé, OntoEdit, and KAON and comparison with OntoXpl [27] will be discussed.

4.1. Protégé and OntoXpl

As one of the most popular ontology and knowledge-base editors, Protégé (version 2.1) provides powerful functions to design and implement ontologies. Protégé lets authors create and import contents. It also allows authors to edit both the original and imported contents [15]. As an ontology information exploration tool, OntoXpl provides enhanced browsing, searching and linking of relevant contents and inferred information in the domain. In order to get all implicit inferences based on the explicit knowledge base, OntoXpl connects and communicates with the OWL reasoner Racer, reorganizes the result information, and provides results to ease authors' understanding of the knowledge.

4.1.1. Loading OWL file

Currently most ontology editors such as Protégé and OilEd can support the Web Ontology Language (OWL). They may have different rules when designing ontology objects. Some may allow naming a concept, role and instance beginning with a number such as "3.5_inches" shown in Figure 4.1, while others may throw an exception when loading the ontology files with different naming convention. OntoXpl is designed to support the OWL naming policy.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:about="3.5_inch">
    </owl:Class>
</rdf:RDF>

```

Figure 4.1: Concept name begins with a number

4.1.2. Implicit axioms

In order to help users understand the ontology file, it would be better that users can see both the explicit defined objects in the domain and the inferred results directly through the GUI.

Let us have a look at an example, in the “*test.owl*” ontology file, users defined concepts *C1*, *C2*, *C3*, a **symmetric** role *OR1*, a **transitive** role *OR2*, and five instances *indi11*, *indi12*, *indi31*, *indi32*, and *indi33* as shown in Figure 4.2. Since “*OR1*” is a symmetric property, users may want to know whether there exists *OR1(indi12, indi11)* or not. The inferred results, however, cannot be displayed correctly through Protégé’s GUI.

In contrast, OntoXpl tells users that the most specific types for “*indi11*” and “*indi12*” are “*C1*” and “*C2*”, as shown in Figure 4.3. By using the Abox queries/browsing functions provided by OntoXpl, users can easily get the inferred results such as “*OR1(indi12, indi11)*”, “*OR2(indi31, indi33)*”, as shown in Figure 4.4.

C1 $\equiv \vee$ OR1 C2	
C2 $\equiv \vee$ OR1 C2	
Indi11 : C1	
Indi12 : C2	
indi31, indi32 : C3	
Symmetric Role OR1	
Transitive Role OR2	
< indi11, indi12 > : OR1	
< indi31, indi32 > : OR2	
< indi32, indi33 > : OR2	

Figure 4.2: “test.owl” designed by Protégé and GUI results

Current Individual is: indi11 [NL Expl Page]	
Name space for this individual is: http://a.com/ontology	
Type: the most-specific atomic concepts of which an individual is an instance	
Type (2) :	<ul style="list-style-type: none"> • <u>C1</u> • <u>C2</u>
Sibling level instances (2) :	<ul style="list-style-type: none"> • <u>indi11</u> type: C2 • <u>indi12</u> type: C2
Ancestor level instances (0) :	• <i>TOP(no instance)</i>
Parents level instances (0) :	• <i>TOP(no instance)</i>
Children level instances (0) :	• <i>BOTTOM(no instance)</i>
Descendant level instances (0) :	• <i>BOTTOM(no instance)</i>
Current Individual is: indi12 [NL Expl Page]	
Name space for this individual is: http://a.com/ontology	
Type: the most-specific atomic concepts of which an individual is an instance	
Type (2) :	<ul style="list-style-type: none"> • <u>C1</u> • <u>C2</u>
Sibling level instances (2) :	<ul style="list-style-type: none"> • <u>indi11</u> type: C2 • <u>indi12</u> type: C2
Ancestor level instances (0) :	• <i>TOP(no instance)</i>
Parents level instances (0) :	• <i>TOP(no instance)</i>
Children level instances (0) :	• <i>BOTTOM(no instance)</i>
Descendant level instances (0) :	• <i>BOTTOM(no instance)</i>

Figure 4.3: Most specific types for “indi1” and “indi2”

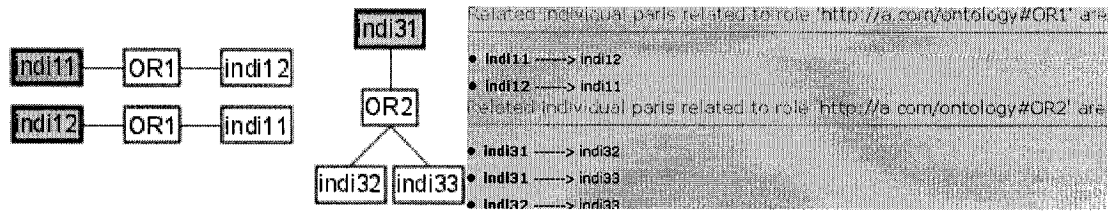


Figure 4.4: Referred results by OntoXpl

In addition, OntoXpl provides summary reports for versatile kinds of properties and concepts such as equivalent concept, inverse functional roles, transitive roles, symmetric roles. Details about Axiom browsing have already been presented in Chapter 3.

4.1.3. Natural language explanation

In Protégé, when the mouse pointer moves over a concept name, a floating layer will be generated with the natural language illustration for the specific concept. However, when the concept is defined with complex asserted conditions, the natural language explanation might become difficult to read.

OntoXpl separates the natural language explanation section into three parts: concepts, roles, and instances. For every named object defined in the domain, its OWL source definition along with its natural language explanation will be illustrated. In contrast to the NL part in Protégé, OntoXpl's NL function is more inclined to illustrate the related information based on the specific objects. Moreover, within the OntoXpl natural language part, all roles, instances, classes defined in the domain will be shown up as functional hyperlinks; detailed information have been presented in section 3.2.

4.1.4. Output results

We separate Protégé's output files into two large groups: the ontology file and the HTML outputs for concepts defined in the domain. The former records how the ontology is designed. The latter is the HTML outputs, which include information such as classes, instances, subclasses, and extended parent classes defined exactly by the system and users. The outputs do not include any inferred or reasoned results. Moreover, Protégé lists all objects defined within the domain without any filter functions, and it takes some time for users to find out information of interest to specific questions. It does not support users to formulate complex queries either.

OntoXpl stresses the importance on both the explicit and implicit information defined in the domain. Through an easily understandable way for users to quickly catch the domain information, OntoXpl provides three kinds of output formats – DIG syntax, space-tree format, and HTML outputs.

First, OntoXpl generates the DIG syntax supported by the Description Logic Implementation Group (DIG) [10]. DIG is effectively an XML Schema for a DL concept language with ask/tell functionality. It will allow users to build plug and play applications where alternative reasoners can be seamlessly interacted with the systems.

The second output format is the space tree supported syntax for concepts, roles and instances. Very often users would like to find out more about a specific object. Therefore,

unrelated information should be excluded as much as possible. On the other hand, users should only see the contents according to specific interests or problems. By communicating with Racer, OntoXpl calculates and transforms the OWL definition of concepts, roles and individuals into a space tree readable XML syntax file. This is useful when there are hundreds of objects, while users are only interested in part of them. By generating the space-tree output hierarchies, users are able to focus on the key information and filter out the less important things. Instead of reading everything one by one, users may be more curious to know how specific instances are related to other instances, and what kinds of roles have been used to setup the relationship between different instance pairs.

As a result, OntoXpl fully supports Abox queries. We formulate three kinds of templates to help users browse detailed individual information. Also for expert users, we provide a way for them to formulate complex queries in a lisp-like notation. Details are in section 3.7.

For instances, when users are interested in “*Jerry*”, OntoXpl will provide only the roles and instances related to “*Jerry*” and filters out the unrelated objects. As shown in Figure 4.5. users know that “*Jerry first appears in the movie ‘Puss gets the Boot’*”, “*has character brown, cheeky, cherubic, happy minding own business until cornered piqued or provoked, impish, look out for other little guys, and small*”, “*in the same cartoon series as muscles, spike, the duckling, tome and Toodles*”, “*is appetizer for Tom*”, “*a cousin of muscles*”, and “*is a friend of spike and the duckling*”. However, not all the information is

explicitly defined in the domain; some of the inferred results are generated by the reasoner and displayed to users. For example, users define “*In_same_cartoon_series*(Jerry, Tom)”, “*In_same_cartoon_series*(Spike, Tom)”, and “*transitive role In_same_cartoon_series*”, the reasoner will let us know that “*In_same_cartoon_series*(Jerry, Spike)”.

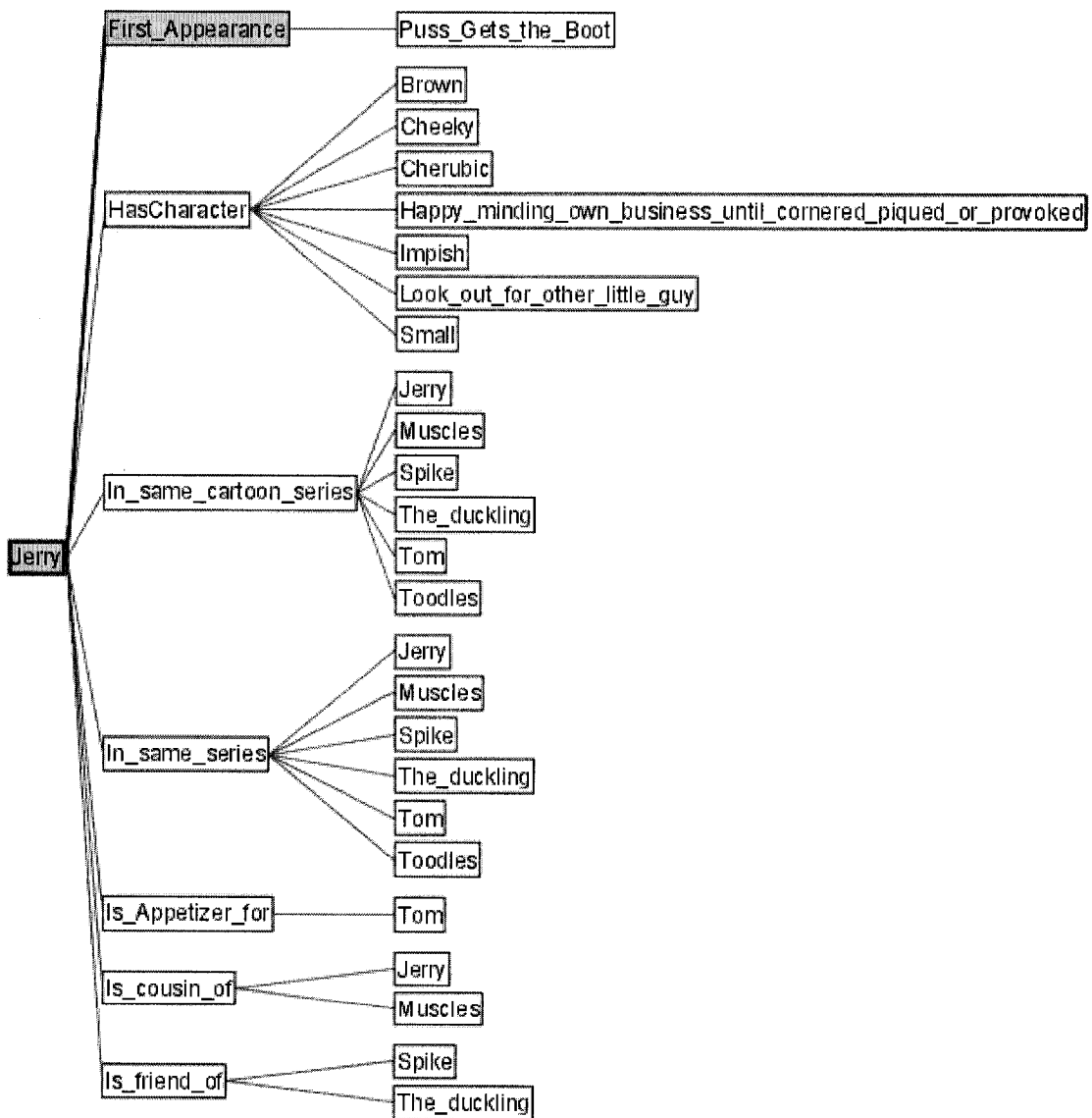


Figure 4.5: Abox space-tree generation result

The last output format provided by OntoXpl is the HTML or JSP output. Since OntoXpl itself is a web-based application, it provides a flexible way to give feedback to users. In addition, all the functions and objects named in the domain have been linked together. Users can easily retrieve information related to a specific object. More details are provided in the following section.

4.1.5. System Architecture

As an open-source pure java application, Protégé provides an extensible architecture for the creation of customized knowledge-based applications. It is often more time-consuming and complex to develop a Swing and java applet GUI. On the other hand, web framework provides a way to create the GUI with less efforts and more flexibility. Besides, many web development tools such as Dreamweaver and FrontPage are available that can make the web development a much easier task.

OntoXpl, as a web application, has a Client/Server architecture. Compared to pure java application, OntoXpl requires less resources at the client side; that is, whenever the HTML browsers have been installed on the computer, it is not necessary that the java environment be setup at the client side as well. In addition, because of the client/server architecture, it enables users' distributing heavy computation tasks to the server side, which is especially useful for less powerful clients such as PDAs and Cell phones.

In addition, since the goal of OntoXpl is to provide a global picture of the knowledge base, a web application is a well-suited tool. Firstly, web is the best tool to link everything together because of the hyperlink function supported by web browsers. Each of the named objects such as the concept, role and instance name is a functional link. Users can easily search all the related information concerning the selected object by opening one or several new windows or viewing the objects within the current page just by clicking the link. Secondly, web browsers help users browse XML syntax files easily. Also, with the support of XSLT, users are able to browse the xml outputs easily in a flexible way. OntoXpl generates the space tree syntactical outputs for concept, role and instances. Users can easily click the “download” link and read the XML syntax file through web browsers. However, pure java application requires a “javax.swing.JTree” object to support showing the XML hierarchy. OntoXpl is a browser-based web application developed based on the web architecture. Users can run OntoXpl and go through all functions within OntoXpl through any popular web browser such as Internet Explore, Netscape, Mozilla, etc.

4.2. OntoEdit

OntoEdit, developed by AIFB, University of Karlsruhe, is an Ontology Engineering Environment that supports the development and maintenance of ontologies using graphical means. The current version of OntoEdit is 2.6.6. OntoEdit is built on top of an internal ontology model. The internal ontology model can be serialized using XML.

OntoEdit provides a quick and intuitive visualization of the definition according to users' design. Within the graph, it shows all roles, instances, and concepts. When ontologies include complex definitions, information will be messed up in the GUI. It is hard for users to find and retrieve only part of the knowledge base such as a specific concept, role, or instance that they are interested in. Besides, the current version does not support connecting to any reasoner. OntoEdit will not give feedbacks or reclassify the hierarchy if there are errors in an ontology design.

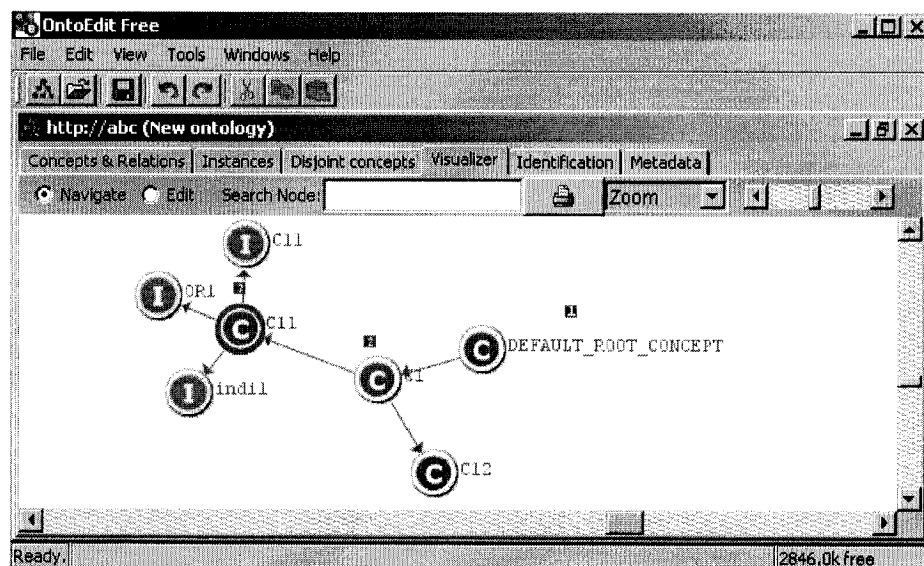


Figure 4.6: OntoEdit knowledge base visualization

OntoEdit only accepts DAML+OIL, Directory import, Excel import, Flogic import, and RDF(s) import. However, it does not support OWL import. OntoEdit supports object name checking partially. As shown in Figure 4.7, users define concept “C11”, role “ORI”, and instances “ind11, ORI and C11”. In addition, we also know that “C11” and

“C12” are disjoint concepts. However, the system allows users to define “*ind1*” as an instance for both concepts, “C11” and “C12”.

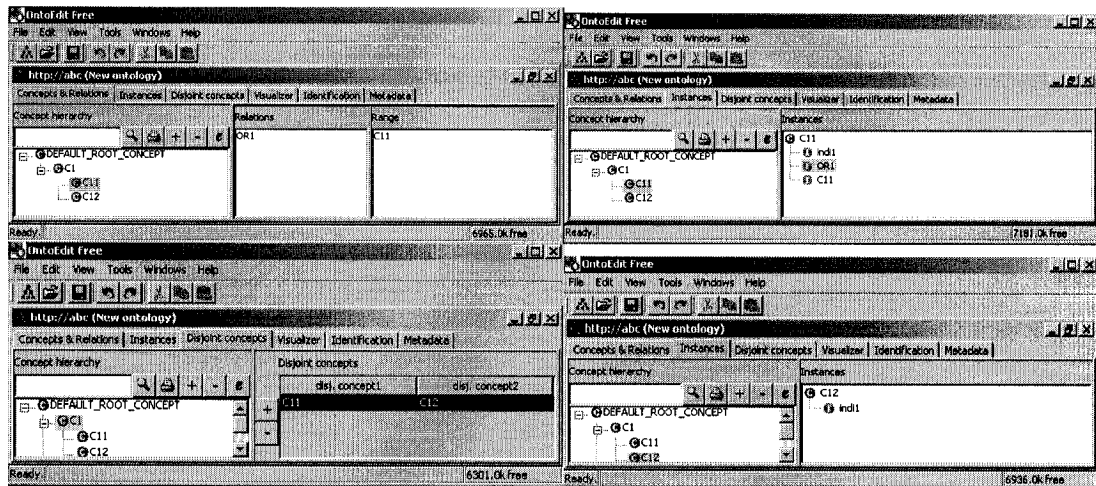


Figure 4.7: OntoEdit GUI

4.3. KAON

KAON is an open-source ontology management infrastructure targeted for business applications. It includes a comprehensive tool suite allowing ontology creation and management and provides a framework for building ontology-based applications. The ontology language of KAON [6] is based on RDF(S) with proprietary extensions for algebraic property characteristics (symmetric, transitive and inverse), cardinality, modularization, meta-modeling, explicit representation of lexical information. It does not support any reasoning results or ABox queries.

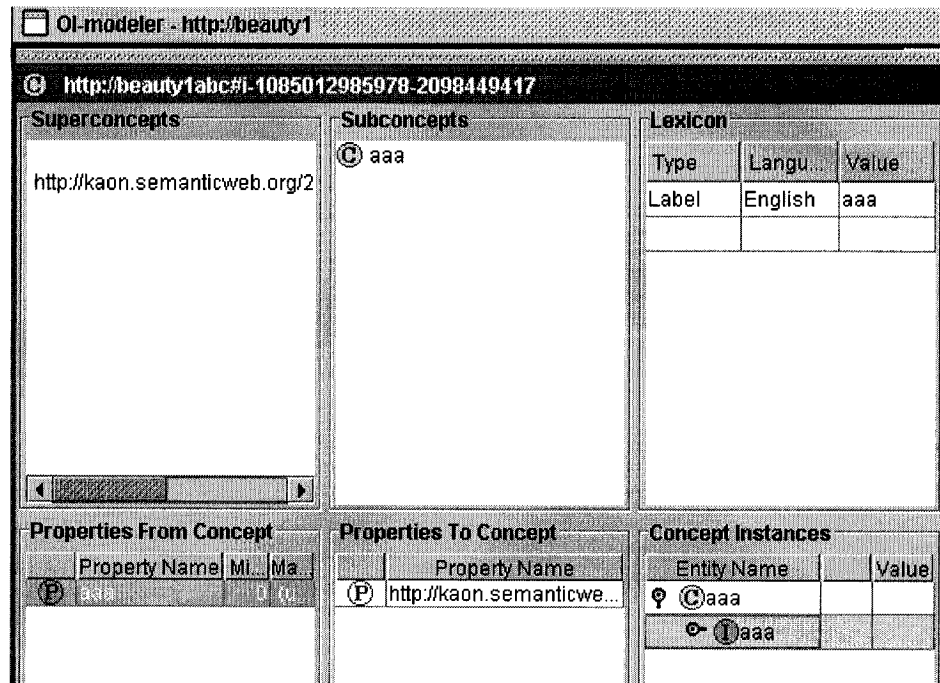


Figure 4.8: KAON main page

4.4. Apollo

Apollo [14] is developed by the Knowledge Media Institute of the Open University. The current version is 1.1 released in June 2003. Apollo supports import and export in OWL syntax. Apollo's modeling is based on the basic primitives, such as classes, instances, functions, relations, etc. The internal model is built as a frame system according to the internal model of the OKBC protocol [23]. Comparing with others, Apollo is the only tool that provides a way for users to specify the name of axioms. However, Apollo does not provide a clear name checking. In Figure 4.9, we can see that concept names, role names and individual names are messed up through the GUI. For example, "OR2" is

defined by Apollo as both instance and role, and “*ind1*” is defined as Axiom, role and instance name.

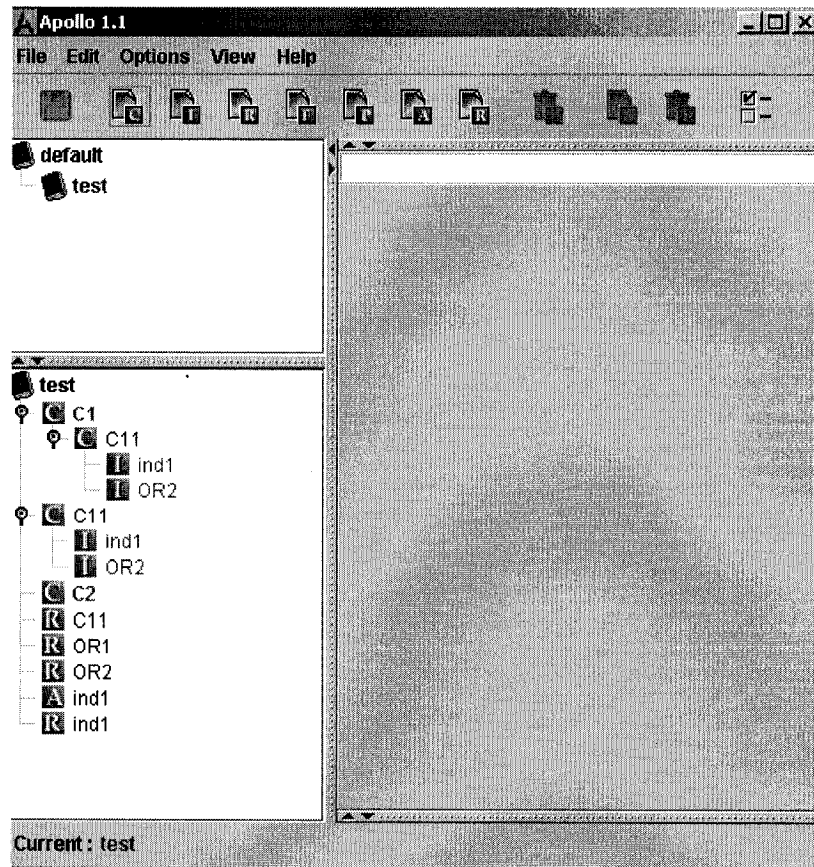


Figure 4.9: Apollo hierarchy information

Apollo does not provide a hierarchy system very well. For instance, suppose *C11* is defined as the only subclass of *C1*, but not the TOP. Apollo will display *C11* as the subclass of *C1*, and at the same level of *C1*. In addition, roles, instances, axioms, concepts are listed in parallel with the same level in the Apollo GUI. As a result, all roles, instances, concepts are “put” together, making it hard to figure out what is what.

4.5. Summary

Convenient and intuitive presentations and manipulations of an ontology interlinking concepts and relations are essential to understand an ontology structure and contents. In addition, because many ontology models support multiple inheritance in concept and relation hierarchies, keeping the associations straight is a challenge. The standard approach is the use of multiple tree views with expanding and contracting levels. A graph presentation is less common, although it can be quite useful for actual ontology editing functions that change concepts and relations. The more effective graph views provide local magnification to facilitate browsing ontologies. Based on the knowledge representation services, Table 4.2 illustrates the main features of Apollo, OilEd, OntoEdit, Protégé, KOAN, and OntoXpl. The features listed are what and how knowledge can be modeled in the tool in order to represent the KR paradigm underlying the knowledge model of the tool, the usability of taxonomies for concepts, roles and relations among instances, the ability to prune the graphs and the possibility to perform zoom into parts of the interesting information. The criteria are listed in Table 4.1.

	Criteria description
Lexical support	Capabilities for lexical referencing of ontology elements and processing lexical content (e.g., searching/filtering ontology terms)
NL explanation	Natural language description for concept, role and instance
Modelling features/limitations	The representational and logical qualities that can be expressed in the built ontology
Web support	Support for Web-compliant ontologies (e.g., URIs, namespace), and {use of the software over the Web} (e.g., browser client) }
Reasoning	Reason inferred results, organize, and display implicit information through GUI
Abox query	Individuals and inferred relationships in the domain; support to formulate complex queries;
Hierarchy Information	<ul style="list-style-type: none"> • <i>Explicit defined and inferred classification of concepts</i> • <i>Role hierarchy</i> information • <i>individual relationship graph</i> - Explicit defined and inferred relationships among instances
Information extraction	Capabilities for ontology-directed capture of target information from content and possibly subsequent elaboration of the ontology
Query	GUI to formulate simple and complex queries based on requirements; also kinds of query templates

Table 4.1: Comparison criteria

	Apollo	Oiled	OntoEdit	Protege	OntoXpl
General description of the tools	A user-friendly knowledge modeling application Version 1.1 http://apollo.op.en.ac.uk	The knowledge model of OilEd is based on that of DAML+OIL Version 3.5 http://oiled.man.ac.uk/	Supporting the development and maintenance of ontologies using graphical means. Version 2.6.6. http://www.ontoknowledge.org/tools/ontoeedit.shtml	An ontology and knowledge-base editor; Open-source java tool with an extensible architecture for the creation of customized knowledge-based applications Version 2.1 http://protege.stanford.edu/	As a supplement of ontology editors, OntoXpl is an information exploration tool Helps users <ul style="list-style-type: none"> • <i>quickly understand the ontology domain</i> • <i>retrieves information</i> • <i>understand the structure and navigates the knowledge-base efficiently.</i>
Lexical Support	No	Limited synonyms	Multiple lexicons via plug-in	WordNet plug-in; wildcard string matching (API only)	The <u>term matching</u> and <u>alphabetical sort</u> . Providing <u>functional links</u> for named objects within domains.
NL explanation	No	No	No	Only for concept. When lots of conditions in one line, NL description might become unreadable messages	Yes. For concepts, roles, and instances in a well organized way
Modeling Features/Limitations	Classes with slots plus relations; functions;	limited XML Schema datatypes; creation metadata; arbitrary expressions as fillers and in constraint axioms; explicit use of quantifiers	limited DAML property constraints and datatypes; no class combinations, equivalent instances	Multiple inheritance concept and relation hierarchies (but single class for instance); instances specification support; constraint axioms input	<ul style="list-style-type: none"> • Browse <u>concept, roles, instances, axioms, & implicit and explicit relations among them in an organized way</u>; • Query templates for objects; • GUI implementation for the New Racer Query Language

	Apollo	OilEd	OntoEdit	Protégé	OntoXpl
Web Support	JAVA pure application	Pure java application RDF URIs; limited namespaces; very limited XML Schema	Resource URIs	<u>Pure java application</u> Limited namespaces; {can run as applet}	<ul style="list-style-type: none"> • <u>Web-based Application</u> • <u>Namespaces</u> for concepts, roles and instances • Objects as functional links • Load ontology from URL e.g. http://www.cs.concordia.ca/cartoon.owl
Reasoning	No	FACT and RACER	No reasoning	RACER	RACER
ABox Query	No	Could input individual name; Provide relationship between instances	No	Yes. individual name search; Provide relationship between instances	<ul style="list-style-type: none"> • ABox templates support - based on instance name, get related instances by limiting some role/instance name/number • Browsing individuals that has relationship with others • ABox inference query results • Spacetime and HTML concept hierarchy with individuals information
Hierarchy Info	Only Concept in JTree format	Only Concept in JTree format	No- All concepts and generated nodes are shown in the same layer- first layer	<u>Concept and Role</u> JTree format	Concept, Role, and individual relationship (indi-role-indi), (role(indi, indi))(concept classification with indi) <ul style="list-style-type: none"> • <i>JTree format</i> • <i>space tree syntax output</i>
Information Extraction	No	Yes. Based on FaCT and RACER, providing concept hierarchy information	No	Yes. Providing the first layer concepts hierarchy information.	Yes. <u>Simple NL parsers; information inference</u> for objects, and related information among them based on RACER
Query	No	Graph query based on FaCT and RACER reasoners	No	Query based on the structure of information saved by users	Interface to support RQL Racer Query Language for complex reasoning queries

Table 4.2: Main features comparison of Apollo, OilEd, OntoEdit, Protégé, KOAN, and OntoXpl

4.6. Interface for Knowledge Retrieval

When a knowledge repository gets very large, finding a particular piece of knowledge can become very difficult. There are two related problems with knowledge retrieval. First, there is the issue of finding knowledge again once it has been stored, understanding the structure of the archive in order to navigate through it efficiently. And second, there is the problem of retrieving the subset of content from the repository that is relevant to a particular problem or subject. This second problem, the dynamic extraction of knowledge from a repository, may well set problems for a knowledge retrieval system that alter regularly and quickly during problem-solving.

In knowledge retrieval, the goal would be to develop user-friendly tools for retrieving knowledge from repositories. One obvious place to begin is to try to exploit natural language. Natural language can be used as the basis for the interface to the knowledge services. For instance, ontologies could be used to interpret knowledge queries in natural language forms from users.

When knowledge is stored or retrieved, it is important to remove any duplication to avoid overloading users with knowledge. OntoXpl is supporting a variety of knowledge fusion techniques to identify and remove such duplications before presenting knowledge to users. For example, we provide three kinds of template and other functions for users to query Abox information. Instead of providing instances all at the same time, only instances related to others have been displayed. Besides, when users want to know more about a specific instance, only information directly related to it will be displayed. For example, there exist “ $R1(indi1, indi2)$, $R2(indi1, indi3)$,

R2(indi2, indi3), R2(indi2, indi4)...R999(indi999, indi1000), and R999(indi999, indi1001)” in an ontology file, and users are interested in “*indi1*”. OntoXpl will display information that are directly related to “*indi1*”, that is, only “*R1(indi1, indi2)* and *R2(indi1, indi3)*” will be displayed instead of the whole graph.

In case users need NL description or more related information such as “*indi2*” displayed in “*indi1*” related information graph, we provide ways for users to change the current object, and display related information based on a new selected object. More detailed examples can be found in Chapters 3 and 4.

For expert users, they may want to retrieve information by formatting complex queries such as the one given in section 3.7. Users are looking for “*a disney mouse, who has nieces, and is a friend of Mickey*”. OntoXpl provides an interface to formulate complex queries in a lisp-like nRQL language [26].

5. Design and implementation

In this section, we discuss about the system architecture of OntoXpl, algorithms and the modules used to support OntoXpl's main features.

5.1. OntoXpl Architecture

5.1.1. System Architecture

OntoXpl system is an ontology knowledge browsing system for a subject-specific collection of information. The subject area is defined by a set of objects and axioms, called ontology. OntoXpl separates content from visualization. The project breaks down into six main modules, shown in Figure 6.1: the OWL ontology file input, user portals or interfaces, Tomcat server, Servlet business-logical analyzing classes, and JAVA API for reasoning results. These modules are partitioned into three groups, those dealing with the Racer reasoning, those dealing with business-logic, and those dealing with displaying information. The reasoning group consists of the TCP-based client for Racer, and all classes based on and related to the Racer reasoning results. The Java Servlets make up the business-logic part, while the style sheet processor, the web server, and the user interface make up the display group. As illustrated in Figure 6.1, ontologies designed by Protégé or other ontology editors will be loaded to the tomcat server through the web portal. Servlets send requirements to the reasoning JAVA API to implement serial reasoning tasks. After that, the results will be sent back to users through the interface. The main techniques used by the OntoXpl system

and sub-system functions are described in the following section. This OntoXpl structure is the modified version based on the requirements and feedback from students in a recent graduate class.

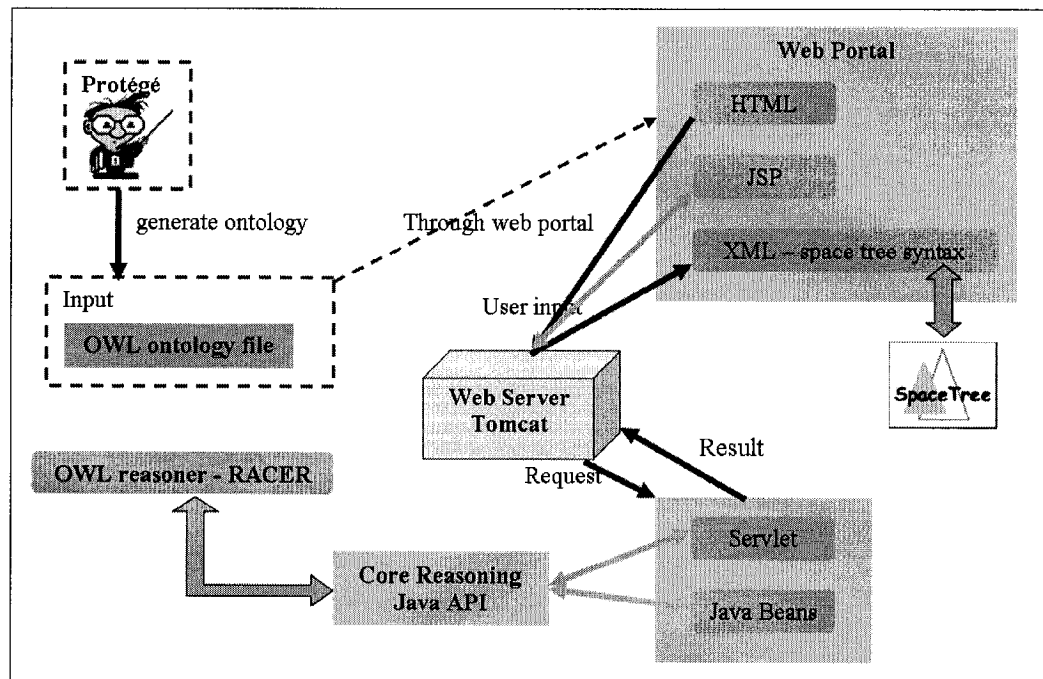


Figure 6.1: The OntoXpl system structure

5.1.2. Servlet vs. JSP

Servlets provide platform-independent, 100% Pure Java server-side modules and fit seamlessly into the framework of the application server. Servlets are compiled into Java bytecode that can be downloaded or shipped across the network; they are truly "Write Once, Run Anywhere." Unlike CGI scripts, Servlets involve no platform-specific consideration or modifications: they are Java application components that are downloaded, on demand, to the part of the system that needs them.

JSP is the Java platform technology for building applications containing dynamic Web content such as HTML, DHTML or XML. A JSP page is a text-based document containing static HTML and dynamic actions that describe how to process a response to the client. At development time, JSPs are very different from Servlets. However, they are precompiled into Servlets at runtime and executed by a JSP engine, installed on a Web-enabled application server such as Tomcat 5.0.

The drawback of this approach is that the creation of the page must be handled in the Java Servlet, which means that if Web page designers want to change the appearance of the page, they would have to edit and recompile the Servlet. With this approach, generating pages with dynamic content still requires some application development experience. Clearly, the Servlet request-handling logic needs to be separated from the page presentation.

The solution is to adopt the Model-View-Controller (MVC) paradigm for building user interfaces. With MVC, the back-end system is OntoXpl's business or logic model, the templates for creating the look and feel of the response is the View, and the code that glues it all together is the Controller. JSPs fit perfectly into this solution as a way of creating a dynamic response or View. Servlets containing logic or managing requests act as the Controller, while the existing business logic rules that implement and realize all the functions of OntoXpl are the Model.

At the beginning of the OntoXpl design phase, only JSPs were used to deal with form actions and complex business model checking and management tasks. Java codes and HTML formatting occur in the same files together. Moreover, JSP grammar such as `<% if ... else... %>` that are in charge of business logics, and HTML codes such as

<layer...>, <a href...>, that should take care of only the layout display are mixed together. This may be easier and quicker to do as a first attempt, but once more and more intricate logic and pages are added into OntoXpl system, the system becomes too large and difficult to maintain.

We then switched to using both Servlet and JSP. The main benefit of using Servlets with JSP, instead of pure JSP, is that the application has a cleaner model, which is much easier to maintain. The Servlets handle all the business logic (access and logic checking, session management, etc), store the resulting data in a Java Bean, and delegate to the proper JSP page.

On the JSP page, we use Java code only when we need it for formatting purposes, such as grabbing data from a bean. Therefore, the tasks of programming and page design have been clearly separated – the programming part deals with the Servlet side issues and are not mixed with the HTML issues. When designing pages, we will not accidentally delete a line of vital code.

Simple JSP works fine for “quick and dirty” web-applications that do not use any complicated logic. In order to achieve the best trade-off, OntoXpl uses Servlet to deal with the business modules and JSP to deal with the presentation. For example, we use Servlets to take HTTP requests from the web portal, generate the request dynamically (possibly querying back-end systems to fulfill the request) and then send a response containing an HTML or XML document to the browser.

5.1.3. Parsers for OntoXpl

In order to analyze the definition of objects defined within the domain, we need to parse the OWL file. As an OWL parser, Jena is used to check whether the physical expression violates any fundamental data relationships. As an OWL Parser, Jena checks the physical expression against an OWL ontology. However, it cannot satisfy the requirements. For example, we are more interested in some kind of JAVA API that can let us know how many and what roles have been used by a concept, what is the definition of a concept, what are the explicit and referred relationships between concepts, roles, and instances. Thus, we moved our attention to XML parsers because OWL follows the XML presentation syntax. OntoXpl uses the Java XML Technology and JRacer API to connect to Racer. In the following sections, we will introduce how Jena and XML Parser such as JAXB, JAXP work.

5.1.4. Jena 2.0 RDF/OWL parser

As a Java framework for building Semantic Web applications, Jena provides a programmatic environment for RDF, RDFS and OWL, including a rule-based inference engine. Jena is open source and has grown out of work from HP Labs Semantic Web Program. The Jena version in the case study is 2.0. In the following sections, when we only mention “parser”, we mean Jena 2.0. Jena has object classes to represent graphs, resources, properties and literals. Now we use examples to illustrate the Jena parsers [1] and writers for RDF/XML (ARP), and N3 RDF triples [20].

First, Jena provides a Java API which can be used to create and manipulate RDF graphs such as the example in Figure 6.2. This example shows how Jena translates information to RDF formats. Also the N3 result is shown in Figure 6.3. Jena dynamically generates new nodes such as the node “*1fc6e42:f77d541d63:-8000*” and illustrates the result in (*Subject, Predicate, object*) format.

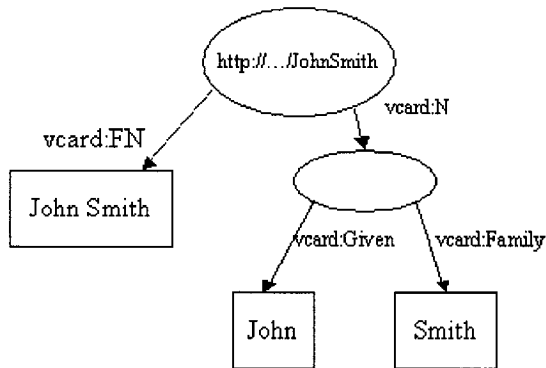


Figure 6.2: Jena parsing Graph result – from http://jena.sourceforge.net/tutorial/RDF_API/figures/fig2.png

```

1fc6e42:f77d541d63:-8000 http://www.w3.org/2001/vcard-rdf/3.0#Family "Smith" .
http://somewhere/JohnSmith http://www.w3.org/2001/vcard-rdf/3.0#FN "John Smith" .
http://somewhere/JohnSmith http://www.w3.org/2001/vcard-rdf/3.0#N 1fc6e42:f77d541d63:-8000 .
1fc6e42:f77d541d63:-8000 http://www.w3.org/2001/vcard-rdf/3.0#Given "John" .

```

Figure 6.3: Jena N3 RDF triple translation results

The second example is an OWL parsing case designed by us. In this OWL file, shown in Figure 6.4, we define two concepts, named ‘r’ and ‘c’, and one role named ‘p’.

owl source file:

```
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
  xmlns:rdfs=http://www.w3.org/2000/01/rdf-schema#
  xmlns:owl=http://www.w3.org/2002/07/owl#
  xml:base="http://www.w3.org/2002/03/owlt/someValuesFrom/premises002">

  <owl:Class rdf:ID="r">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#p"/>
        <owl:someValuesFrom rdf:resource="#c"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:ObjectProperty rdf:ID="p"/>
  <owl:Class rdf:ID="c"/>
</rdf:RDF>
```

Figure 6.4: OWL example for Jena parser

The RDF output result generated by Jena is shown in Figure 6.5. As can be seen, the type of object “r” is class. “r” is the subclass of concept “A0”, whose definition is “*at least one (or more than one) of its instances is(are) concept c for role p*”. Figure 6.6 shows the translation result of the N3 triple.

```
<rdf:RDF xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
  xmlns:rdfs=http://www.w3.org/2000/01/rdf-schema#
  xmlns:owl="http://www.w3.org/2002/07/owl#" >
<rdf:Description rdf:about="http://www.w3.org/2002/03/owlt/someValuesFrom/premises002#r">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
  <rdfs:subClassOf rdf:nodeID="A0"/>
</rdf:Description>

<rdf:Description rdf:nodeID="A0">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Restriction"/>
  <owl:onProperty rdf:resource="http://www.w3.org/2002/03/owlt/someValuesFrom/premises002#p"/>
  <owl:someValuesFrom rdf:resource="http://www.w3.org/2002/03/owlt/someValuesFrom/premises002#c"/>
</rdf:Description>
<rdf:Description rdf:about="http://www.w3.org/2002/03/owlt/someValuesFrom/premises002#p">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
</rdf:Description>
<rdf:Description rdf:about="http://www.w3.org/2002/03/owlt/someValuesFrom/premises002#c">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
</rdf:Description>
</rdf:RDF>
```

Figure 6.5: Jena RDF result

```

Subject:      dc57db:f77df780b8:-8000
Predicate:    http://www.w3.org/2002/07/owl#someValuesFrom
Object:       http://www.w3.org/2002/03owl/someValuesFrom/premises002#c
.
Subject:      http://www.w3.org/2002/03owl/someValuesFrom/premises002#r
Predicate:    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
Object:       http://www.w3.org/2002/07/owl#Class
.
Subject:      http://www.w3.org/2002/03owl/someValuesFrom/premises002#r
Predicate:    http://www.w3.org/2000/01/rdf-schema#subClassOf
Object:       dc57db:f77df780b8:-8000
.
Subject:      http://www.w3.org/2002/03owl/someValuesFrom/premises002#c
Predicate:    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
Object:       http://www.w3.org/2002/07/owl#Class
.
Subject:      dc57db:f77df780b8:-8000
Predicate:    http://www.w3.org/2002/07/owl#onProperty
Object:       http://www.w3.org/2002/03owl/someValuesFrom/premises002#p
.
Subject:      dc57db:f77df780b8:-8000
Predicate:    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
Object:       http://www.w3.org/2002/07/owl#Restriction
.
Subject:      http://www.w3.org/2002/03owl/someValuesFrom/premises002#p
Predicate:    http://www.w3.org/1999/02/22-rdf-syntax-ns#type
Object:       http://www.w3.org/2002/07/owl#ObjectProperty

```

Figure 6.6: Jena N3 result

5.1.5. XML Parser - SAXParser

XML (Extensible Markup Language), a markup language that makes data portable, is a key technology in addressing the ability of parties to communicate and share legacy data with each other even if they are using different information systems. As a result, XML is increasingly being used for enterprise integration applications, both in tightly coupled and loosely coupled systems. The OWL language uses the XML syntax. In order to analyze the syntax of an OWL file, we need to be able to parse the XML file. In the development, we used JAVA SAXParser technology [29] to develop OntoXpl.

The Java APIs for XML falls into two broad categories: those that deal directly with processing XML documents and those that deal with procedures.

- **Document-oriented –**

1. Java API for XML Processing (JAXP) -- processes XML documents using various parsers
2. Java Architecture for XML Binding (JAXB) -- processes XML documents using schema-derived JavaBeans component classes

- **Procedure-oriented (Mainly used by Semantic Web Services)**

1. Java API for XML-based RPC (JAX-RPC) -- sends SOAP method calls to remote parties over the Internet and receives the results
2. Java API for XML Messaging (JAXM) -- sends SOAP messages over the Internet in a standard way
3. Java API for XML Registries (JAXR) -- provides a standard way to access business registries and share information.

The basic outline of the SAX parsing APIs are shown in Figure 6.7. To start the process, an instance of the SAXParserFactory class is used to generate an instance of the parser.

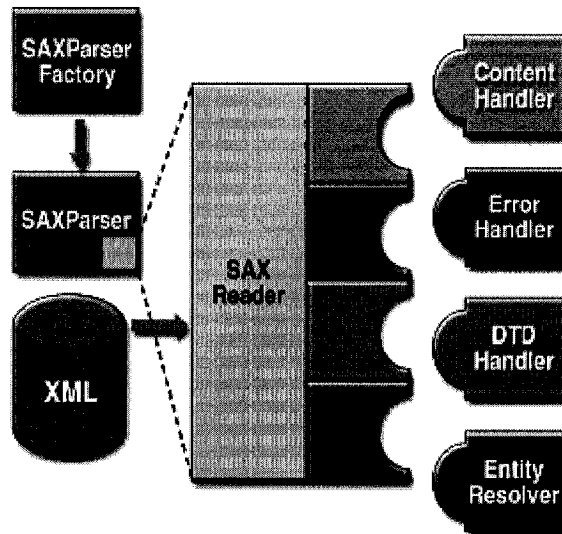


Figure 6.7: SAX APIs

Let us take OntoXpl’s Natural language function as an example, which is developed based on the SAX Parser. As shown in Figure 6.8, all classes under “information.NL” package have a relationship with `javax.xml.parsers` and `org.xml.sax`. Based on the SAX API and OWL grammar, OntoXpl is able to check all concepts, roles, and instances that have been defined within a domain. For example, we defined `<owl:Class rdf:ID="Cartoon_cat">`. In our programs, there exists a method named “`startElement(String namespaceURI, String sName, String qName, Attributes attrs)`”, by which we are able to check the current tag and dispatch functions to roles, concepts, or instances. In order to treat the definition of a specific concept, role or instance as a unity, we need to parse the definition of a specific object, generate temporal xml syntax files, and parse the generated xml files line by line until the end of the tag. The detailed NL parsing file structures are shown in Figure 6.9.

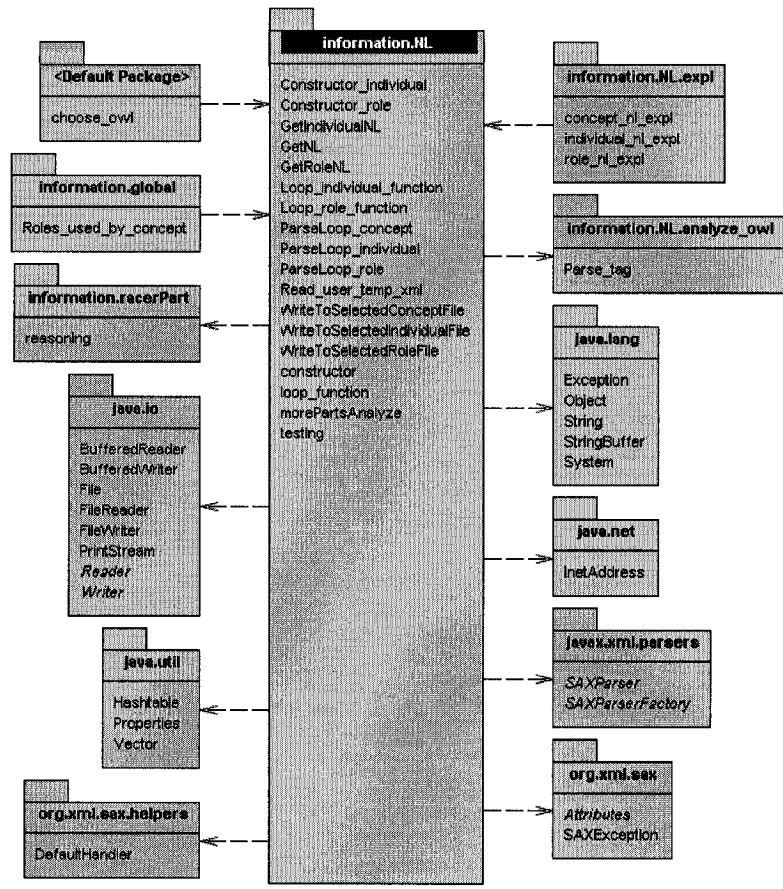


Figure 6.8: NL Package API Info

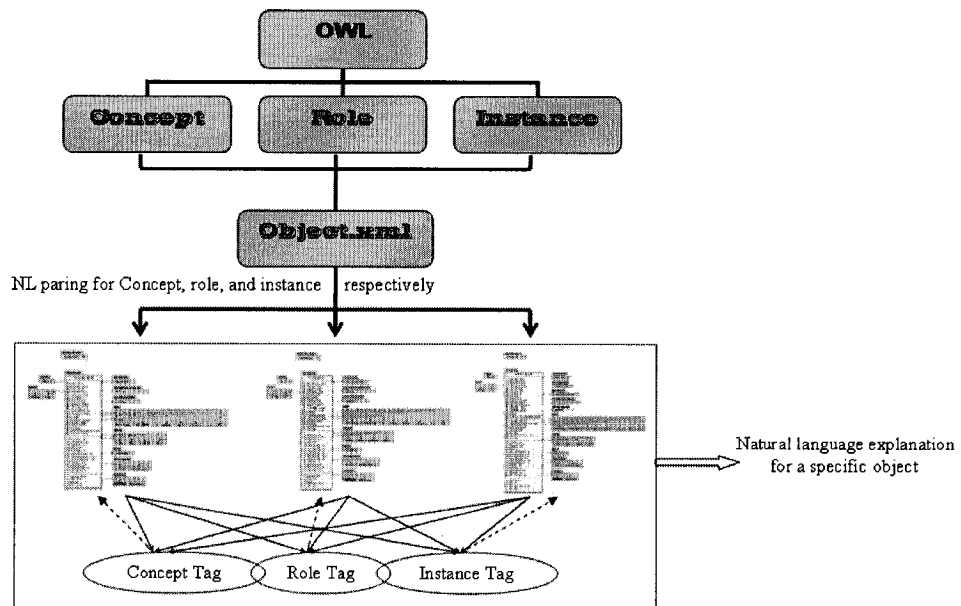


Figure 6.9: NL Parsing Algorithm

5.1.6. Saving knowledge to database vs. dynamically querying results without saving

At the beginning of OntoXpl's design, we saved all concepts, roles, and instances into databases (shown in Figure 6.10). We also designed a user login system to manage login users. In addition, according to login users and ontology files loaded, we saved the ontology file and related those ontology files to login users. Thus, concepts, roles, instances, ontology files and login users had been connected with each other. The advantage of this design is that we are able to manage different users' ontology files. However, there are two disadvantages in this design. When the ontology file is very large, it takes some time to deal with the database operations. The other problem is that it takes lots of disk space to save the ontology file information, concepts, roles, and instances into the database for all users. We assumed that ontology files and their related information will be deleted from databases after users logout from the OntoXpl system; that is why we designed the login system. However, if a user fails to logout, his or her data would be stored in our databases until connections setup by users are lost due to a time out.

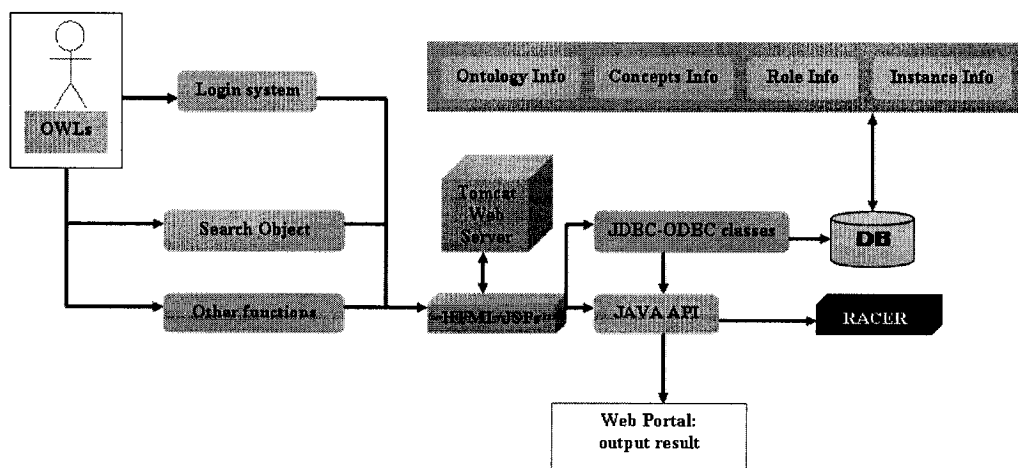


Figure 6.10: Database operation architecture

By using the structure in Figure 6.10, every time we want to select or search for a concept, we need a database operation, where we still need Java API support to get the final results. Using databases to save object information is helpful when we want to record and trace the information loaded by login users. However, for OntoXpl, saving all users login information into the database is only a media; that is, the user management function is not our goal. In addition, it takes some time to perform read/write database operations. Thus, based on the example model, we changed the structure to use only JAVA API and Servlets to realize functions that OntoXpl provides. The modified system structure already shown in Figure 6.1 provides a more clear and efficient structure. However, we do not support multi-users' operation currently. This is left for future work.

5.2. Modules and relationship

In order to realize the functions provided by OntoXpl mentioned in Chapter 3, we have designed OntoXpl based on the system architecture in Figure 6.1. In this section, we explain how the main modules and functions were developed and organized in order to implement OntoXpl's features.

OntoXpl uses MVC modules: Servlet, JSP, and java API. Under each part, we organize our module by functions. For example, we provide "natural language description", "information inference", "Axiom browsing", "Abox functions", and so on. Under Servlet, and java API parts, we organize our modules as shown in Figure 6.11. We will discuss the algorithm and techniques we used for the main functions provided by OntoXpl in the following sections.

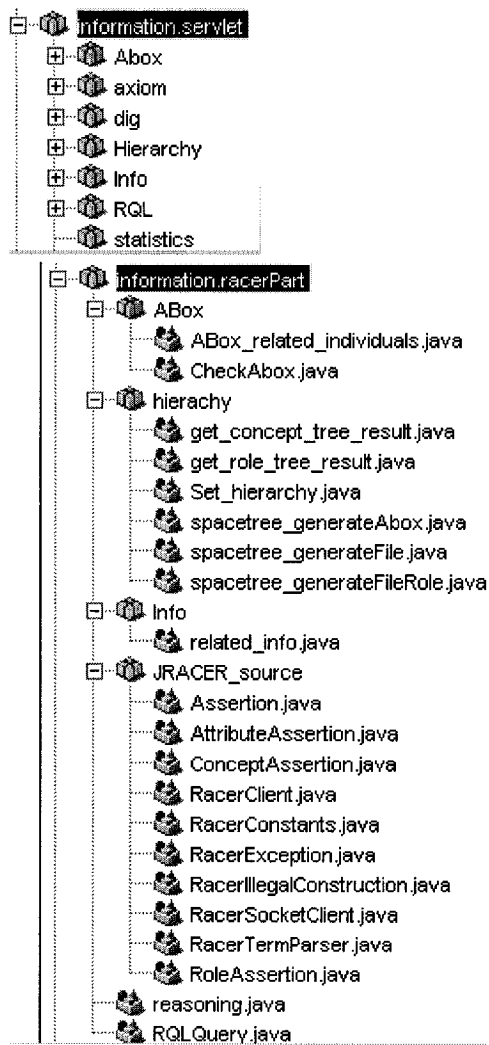


Figure 6.11: Servlet and reasoning API modules

5.2.1. Natural language description

Figure 6.12 provides the basic idea of the natural language description function. Based on the object name, we parse the source file to get the OWL definition and generate the natural language description. Since the object definition can be scattered among the whole ontology file, to organize and combine the exact OWL definition for a specific concept, we parse the source ontology file based on the SAXParser.

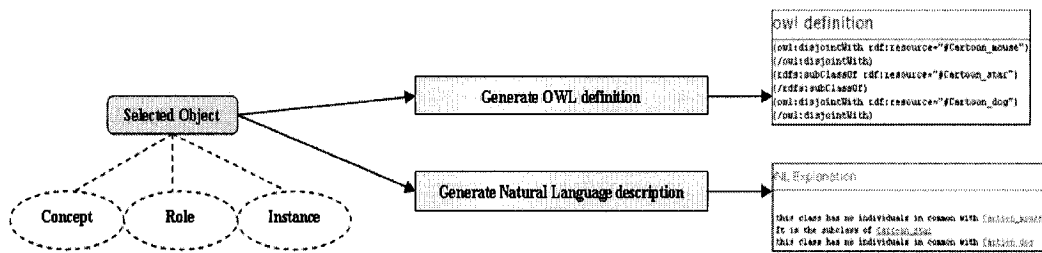


Figure 6.12: Natural language description function schema overview

The algorithm to parse and generate the selected concept's OWL source definition is shown in Figure 6.13. Based on the SAXParser supports, we analyze the source ontology file. We begin from the xml syntax begin tag such as `<owl:Class...>`, check whether the current object is the one that has been selected. If yes, we append the xml begin tag and its related attributes into a string named *"selectConceptOWLDef"*. Next, we check the contents between the begin tag and end tag such as `<owl:Class rdf:ID="Cartoon_cat">XXX</owl:Class>`, if the current concept is *"Cartoon_cat"*, we will append XXX into *"selectConceptOWLDef"*. Last, we check the end element of the xml tag, if it is not the end of the ontology file, we add the end tag definition to the *"selectConceptOWLDef"* string. Until it is the end of the ontology file, we will write the definition recorded in *"selectConceptOWLDef"* into a xml file.

```

checking OS system
string selectedConceptOWLDef := "";
boolean findSelectedConcept := false;
if (windows)
    Setup temporal output file "c:\temp\temp.xml" ;
else if (UNIX, Linux, Mac)
    setup temporal output file "/tmp/temp.xml";
writing xml titles into the temporal output file;
while not the end of XML tag checking
    check startElement begin
        take the next tag of source ontology file
        if (satisfy the searching)
            selectedConceptOWLDef += xml tag begin and related attributes;
            findSelectedConcept := true;
        else
            begin to check characters
    check characters begin
        if(findSelectedConcept)
            selectedConceptOWLDef += xml contents between the tag;
        else
            begin to check endElement
    check endElement begin
        if (satisfy the searching)
            selectedConceptOWLDef += xml tag end and related attributes;
end of XML tag checking
write selectedConceptOWLDef into temporal output file

```

Figure 6.13: OWL source definition generation algorithm

After the OWL definition for a specific object has been generated, we can begin to analyze and generate the natural language explanation for the object. An object can be a concept, role, or instance. As shown in Figure 6.14, there are two main parsers. The first one is in charge of parsing the loaded xml file with all OWL definition for an object layer by layer. For example, there is a concept defined as in Figure 6.15. Parser one will generate five layers. Based on these five layers, the second parser will generate the related NL description. SAXParser provides a good way for us to parse the xml syntax structure and generate the natural language description based on the OWL tags in a well-formatted way.

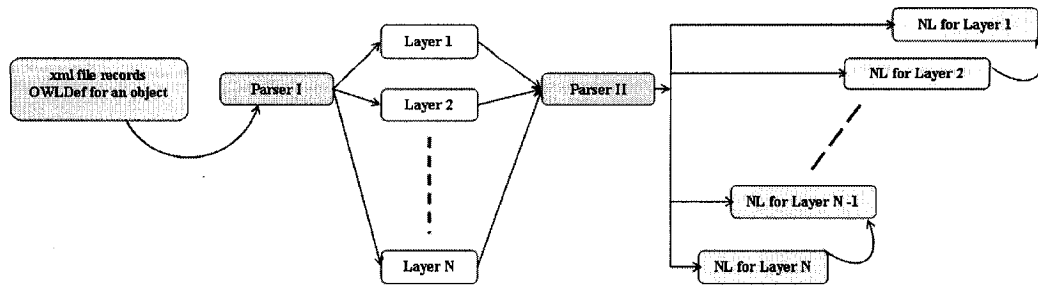


Figure 6.14: Natural language translation flowchart

```

<owl:Class rdf:ID="X1">                                1
  <rdfs:subClassOf>                                    2
    <owl:Restriction>                                  3
      <owl:onProperty>                                  4
        <owl:ObjectProperty rdf:ID="OR1"/>             5
      </owl:onProperty>                                4
      <owl:allValuesFrom>                               4
        <owl:Class rdf:ID="Y"/>                       5
      </owl:allValuesFrom>                             4
    </owl:Restriction>                                 3
  </rdfs:subClassOf>                                  2
  <rdfs:subClassOf>                                    2
    <owl:Class rdf:ID="X"/>                            3
  </rdfs:subClassOf>                                  2
</owl:Class>                                          1

```

Figure 6.15: OWL definition for example concept "X1"

5.2.2. Reasoning system based on Racer

In order to provide reasoning results, reasoning commands [24] are sent to the Racer system via the TCP-based client interface. After parsing the feedback string from Racer, we save the results that are valid object names defined within a domain into a vector or a hashtable. Servlet will call the java API reasoning function, read, organize

and send the reasoning results to users. In a way, Servlets, as the business model controller, take care of the combining calling of the reasoning functions provided by java APIs.

```
setup racer connection (ip, port)
send racer command
    (concept-children CN)
    (concept-parents CN)
    (concept-ancestors CN)
    (concept-descendants CN)
    (concept-instances CN) ...
use StringTokenizer to parse racer results
get rid of "INV", "NIL", "%&" and other unrelated information generated by Racer dynamically
save parsed results into Vector or Hashtable
```

Figure 6.16: Algorithm for reasoning results based on Racer

Two UML example graphs in Figure 6.17 and 6.18 display the relationship between `information.servlet` package and the `reasoning.java` class.

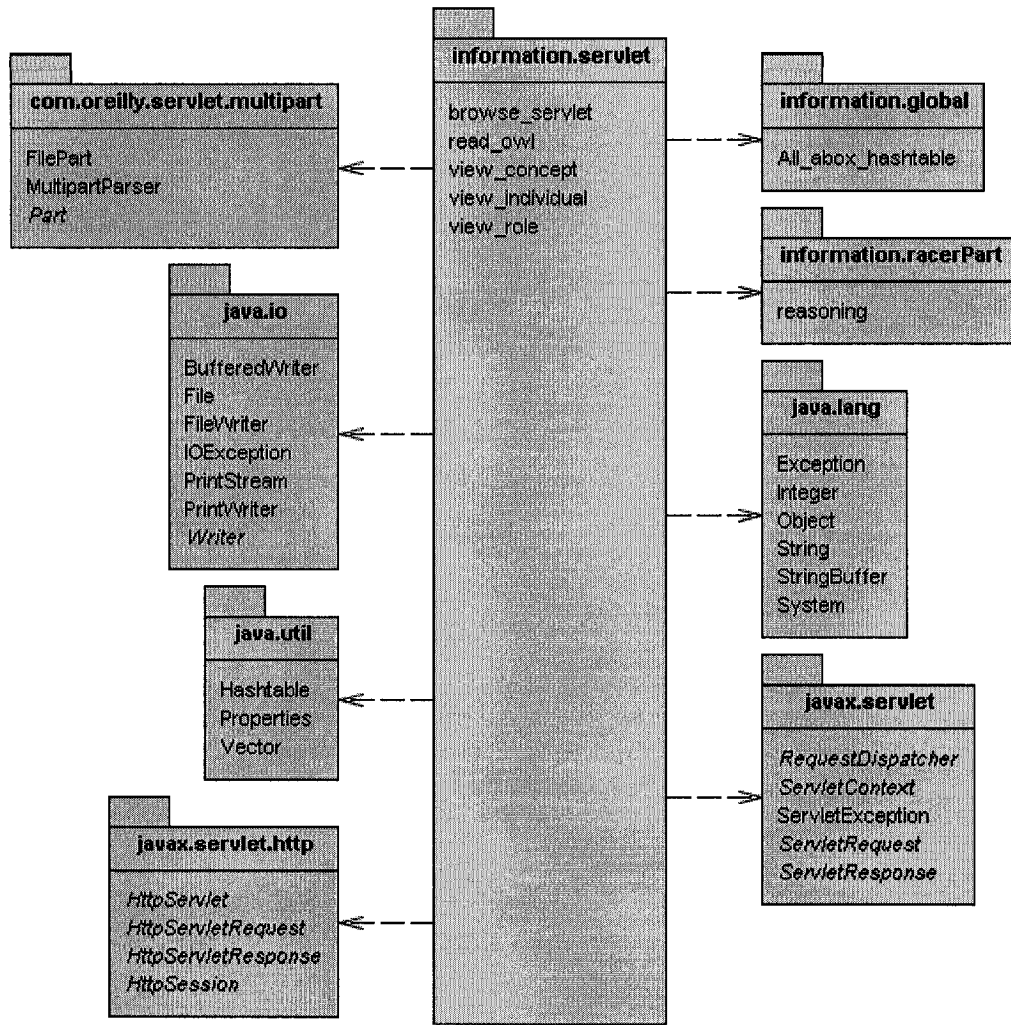


Figure 6.17: UML package information for "information.servlet" package

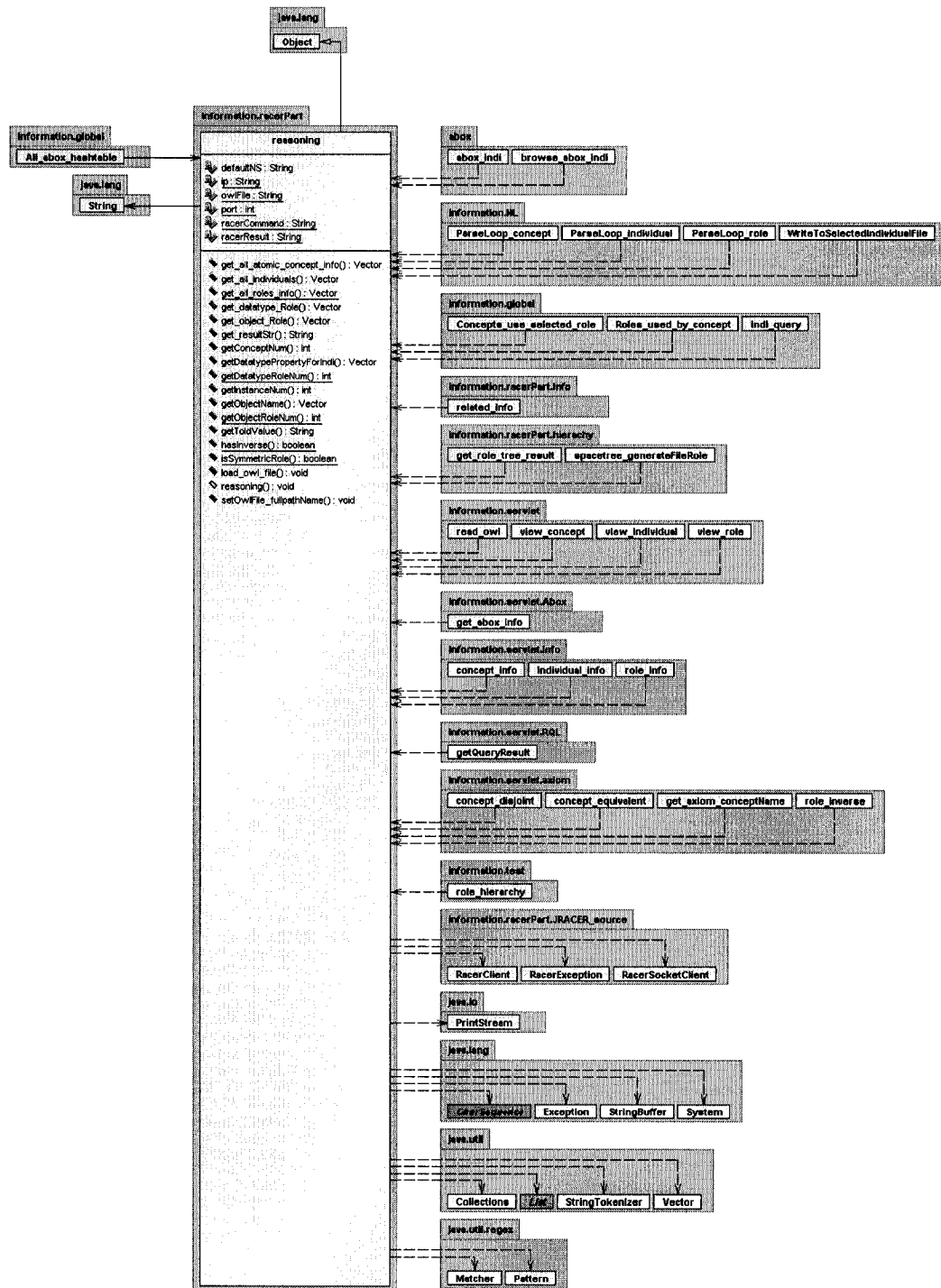


Figure 6.18: UML package information for reasoning.java class

5.2.3. Spacetree output

Though the current version of Spacetree syntax does not support true graphs, it still provides us a good way to display concept hierarchies, role hierarchies, and instance relationships as a tree structure. The general idea to generate Spacetree syntax outputs based on an ontology source file is shown in Figure 6.19.

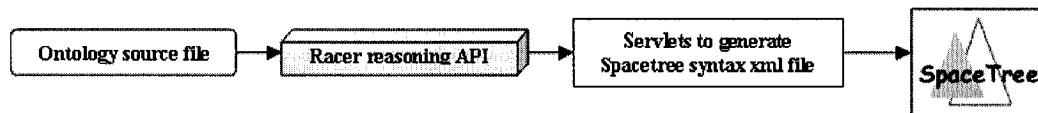


Figure 6.19: Spacetree output generation

The grammar of Spacetree is based on the indents of the combination of the tag pair “<node></node>”. For example, there is one ontology file as defined in Figure 6.20. The Spacetree syntax result is shown in Figure 6.21. Based on the reasoning results from the Racer reasoning API developed by OntoXpl, Figure 6.22 provides the algorithm to generate the Spacetree output for concept hierarchy information. The Spacetree output for role hierarchy information is similar. However, the relationship for instances Spacetree output is different. Figure 6.23 displays the algorithm for individual relationship outputs.


```

<?xml version="1.0"?>
<rdf:RDF
  xmlns="http://a.com/ontology#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://a.com/ontology">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="C2">
    <owl:equivalentClass>
      <owl:Class rdf:about="#C1"/>
    </owl:equivalentClass>
  </owl:Class>
  <owl:Class rdf:ID="C1">
    <owl:equivalentClass rdf:resource="#C2"/>
  </owl:Class>
  <owl:Class rdf:ID="C11">
    <rdfs:subClassOf rdf:resource="#C1"/>
  </owl:Class>
  <owl:ObjectProperty rdf:ID="OR1"/>
</rdf:RDF>

```

Figure 6.20: example ontology source file

```

<node>TOP
  <node>C1
    <node>C11</node>
  </node>
  <node>C2
    <node>C11</node>
  </node>
</node>

```

Figure 6.21: Spacetree syntax output file

```

1: set "Top" concept
2:   set <node> begin tag for "Top"
3:   get hierarchy information based on (current concept name)
4:     if (no sub-concept)
5:       write end tag </node>
6:       return;
7:     else
8:       if(not dup information)
9:         for(i :=0; i<total sub-concept number; i++)
10:          write begin tag <node> for current sub-concept
11:          goto 3 (based on the current sub-concept)
12:        end for
13:      end if
14:    end if
15:  end of get hierarchy information
16:  set </node> for "Top"

```

Figure 6.22: Spacetree concept hierarchy output generation algorithm

```

1: write begin tag <node> for user selected individual
2:   get all roles related to (current individual)
3:   if(no related Role)
4:     write end tag </node>
5:     return
6:   else
7:     Vector relatedRoleVect saves all related roles
8:     for(i:=0; i<relatedRoleVect.size; i++)
9:       String currentRole := relatedRoleVect.get(i)
10:      write begin tag <node> for currentRole
11:      get all individuals related to current individual and currentRole
12:      Vector relatedIndiVect saves all individuals related to current individual and currentRole
13:      for(j:=0; j< relatedIndiVect.size; j++)
14:        currentIndividual := relatedIndiVect.get(j)
15:        write begin tag <node> for currentIndividual
16:        goto 2 using (currentIndividual)
17:      end for
18:    end for
19:  end if
20:  end of getting related roles
21: write end tag for user selected individual

```

Figure 6.23: Spacetree individual relationship output generation algorithm

6. Evaluation

In a recent graduate class COMP 691B: Foundations of the Semantic Web, there were about 40 students divided into 16 groups. Each group was assigned to design a different ontology. The 16 different ontologies' domains were: Clothing, Diet, Storage Area Network (SAN), Watch, Skincare Products, Ontology for Web Application Patterns (OWAP), Writing tools, Pharmaceutical company, Ski equipments, Programming languages, Animals, Lodging, Cellular phones, University Websites, Music, Refrigerator. The average numbers per each ontology file for concepts, roles and instances were 200, 40, and 100, respectively.

Students used Protégé, the ontology editor, to design and implement their ontology files, which are the input for OntoXpl (shown in Figure 6.1). Since each member of a group contributed in the development of their ontology, it was important for them to understand each other's tasks and processes at different steps. According to students, OntoXpl helped them understand the objects they created and modified among group members. Besides, OntoXpl helped students fix confusing issues such as missing namespaces, NULL pointers exception for "open NL pages", installation of the OntoXpl system on a directory different from ROOT, the support of running RACER on a different machine, the function to send ontology file from online, and so on. The feedback from students helped us improve the functionalities of OntoXpl and its interface.

7. Conclusions and Future work

In this chapter, we provide a summary of our work on developing OntoXpl, and highlight its features. Concluding remarks and future directions are also provided.

7.1. Conclusions

We have introduced the OntoXpl system, a first step toward an OWL ontology exploration tool designed to browse knowledge bases and explore information contained in them. OntoXpl is intended to complement ontology editors or other ontology visualization tools. The OntoXpl system supports navigation and exploration of information for ontology-based application development. For example, OntoXpl helps extract the OWL structure supported by W3C, such as functional properties, inverse roles, transitive roles, symmetric roles, and so on. In order to avoid providing literally everything to users, OntoXpl can help narrow down and retrieve only the relevant information, organizes and displays desired information in a manner convenient to users.

Also, OntoXpl can be used as a tool to evaluate existing ontology applications. For example, to help users decide whether to create new ontologies or reuse the existing ones, based on the structure of the information about roles, concepts, instances, axioms, etc.

In addition, OntoXpl provides full functional supports for OWL and Aboxes. For instance, there are three kinds of templates to query Abox information, spacetree

syntax outputs of relationships between instances, and the GUI for expert users to formulate complex queries.

In a recent experiment of evaluating OntoXpl, 40 graduate students in a course on Semantic web, were assigned to design and implement 16 different OWL ontologies of several hundred concept names, demonstrated that OntoXpl provided helpful information to users about ontologies that was otherwise not easily available using ontology editors such as Protégé or OilEd.

7.2. Future work

There are three main items that are in the to do list. First, we plan to implement the statistics function, a feature to be realized at the right hand side of the main page. This will include information on the ontology imported, such as how many ontology files have been imported and what are they. It also includes information on the concepts, roles, instances used. For example, what are atomic concepts, how many concepts have only necessary definition, how many concepts have both necessary and sufficient definition. The second item is optimization for larger ontologies containing thousands of concept names. We are able to ease the communication load with Racer through the TCP protocol by optimizing the JAVA API reasoning component. For instance, instead of sending and retrieving the results through Racer every time, we can save the query results into a hash table, vector or an array for most regular queries. Examples of regular queries include “all-atomic-concepts”, “all-roles”, and “all-individuals”. Finally, in order to allow multi-users manage their loaded ontologies, OntoXpl needs a user login mechanism to manage the login users and

their loaded ontologies. This would allow users to go through different Tbox and Abox information at the same time. In addition, this would allow users to compare two or more loaded ontology files. For example, when there are two similar domain ontology files, an extended OntoXpl could help users to identify main differences among concepts, roles, instances, axioms, and so on.

8. References

- [1] Brian McBride. Jena 2 - A Semantic Web Framework. Hewlett-Packard Laboratories, Bristol, UK. <http://www.hpl.hp.com/semweb/jena.htm>;
- [2] Clyde W. Holsapple, K. D. Joshi. A Collaborative Approach to Ontology Design 2002 ACM 0002-0782/02/0200;
- [3] C. Welty and B. Smith (eds.). *Formal Ontology and Information Systems*. New York, ACM Press, 2001;
- [4] D. Brickley and R.V. Guha. RDF vocabulary description language 1.0: RDF Schema, <http://www.w3.org/tr/2002/wd-rdf-schema-20020430/>, 2002;
- [5] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview (W3C Recommendation 10 February 2004);
- [6] D. Oberle, R. Volz, B. Motik, and S. Staab. An extensible ontology software environment. In *Handbook on Ontologies, International Handbooks on Information Systems*, chapter III, pages 311–333. Steffen Staab and Rudi Studer, Eds., Springer, 2004;
- [7] Dublin Core 2004 (<http://dublincore.org>);
- [8] F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. *OWL web ontology language reference*;
- [9] Gruber, T.R. Toward principles for the design of ontologies used for knowledge sharing. *Int. J. Hum. Comput. Stud.* 43, 5/6 (1995), 907–928;
- [10] Ian Dickinson. Implementation experience with the DIG 1.1 specification. HP Labs Bristol. 10th May, 2004;
- [11] International Organization for Standardization (ISO/IEC 11179-1) 1998;

- [12] Jeff Heflin, James Hendler, and Sean Luke. SHOE: A Knowledge Representation Language for Internet Applications, 1999;
- [13] J. Grosjean, C. Plaisant, and B. Bederson. Spacetree: Supporting exploration in large node link tree, design evolution and empirical evaluation. In Proceedings of IEEE Symposium on Information Visualization, pages 57–64, Boston, USA, October 2002;
- [14] Matej Koss. Apollo – the user guide(describes version 1.0) 2001–2002 Knowledge Media Institute, The Open University, UK.;
- [15] N. F. Noy, M. Sintek, S. Decker, M. Crubezy, R. W. Fergerson, and M. A. Musen. Creating semantic web contents with Protege-2000. IEEE Intelligent Systems, 16(2):60–71, 2001;
- [16] O. Lassila and R.R. Swick. Resource description framework (RDF) model and syntax specification. recommendation, W3C, February 1999;
- [17] R. Möller, R. Cornet, and V. Haarslev. Graphical interfaces for Racer: querying DAML+OIL and RDF documents. In Proc. International Workshop on Description Logics – DL’03, 2003;
- [18] S. Bechhofer, I. Horrocks, and C. Goble. OilEd: a reason-able ontology editor for the semantic web. In Proceedings of KI2001, Joint German/Austrian conference on Artificial Intelligence, September 19-21, Vienna. LNAI Vol. 2174, Springer-Verlag, 2001;
- [19] The DARPA Agent Markup Language Homepage <http://www.daml.org/>;
- [20] Tim Berners-Lee, Dan Connolly, and Sandro Hawke. Semantic Web Tutorial Using N3. May 20, 2003. <http://willware.net:8080/cwm-tutorial.pdf>;
- [21] Tim Berners-Lee, James Hendler and Ora Lassila, “The Semantic Web - A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities”. May 2001;

- [22] T. Liebig and O. Noppens. OntoTrack: Fast browsing and easy editing of large ontologies. In Proceedings of The Second International Workshop on Evaluation of Ontology-based Tools (EON2003), located at ISWC03, Sanibel Island, USA, October 2003;
- [23] Vinay K. Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp, and James P. Rice. OKBC: A Programmatic Foundation for Knowledge Base Interoperability. In Proceedings of AAAI-98, July 26-30, Madison, WI;
- [24] Volker Haarslev and Ralf Möller. Racer system description. In *Proc. Of the Int. Joint Conf. On Automated Reasoning (IJCAR 2001)*, 2001;
- [25] Volker Haarslev and Ralf Möller. Optimization techniques for retrieving resources described in OWL/RDF documents: First results. In Proceedings of the Ninth International Conference on the Principles of Knowledge Representation and Reasoning (KR2004), June 2004;
- [26] Volker Haarslev, Ralf Möller, R. Van Der Straeten, and M. Wessel. Extended query facilities for Racer and an application to software-engineering problems. In Proceedings of the International Workshop on Description Logics (DL-2004), Whistler, BC, Canada, June 2004;
- [27] Volker Haarslev and Ying Lu and Nematollah Shiri. OntoXpl Exploration of OWL Ontologies. *International Workshop on Description Logics*, 2004;
- [28] Y. Sure, J. Angele, and S. Staab. OntoEdit: multifaceted inferencing for ontology engineering. *Journal on Data Semantics*, 2800/2003:128–152, 2003;
- [29] Web Services Made Easier – The Java™ APIs and Architectures for XML. A Technical White Paper. Sun Microsystems, Inc. JUNE 2002 Revision 3.

Appendix

A.1 Example about loading OWL ontology into OntoXpl

Assume we have tomcat and java already installed successfully on the machine running Windows XP. After downloading Racer to the local disk, users need to open a DOS window, go to the directory saving Racer, and type “Racer -http port number” to start Racer. For example, “Racer -http 9999”. This causes the system to use http protocol and port 9999 to communicate with Racer. According to users’ requirements, typing “Racer -http 0”, users can also close the http protocol provided by Racer. We then open another DOS window, go to “Tomcat_directory/bin/”, type “startup” under windows operating system or “startup.sh” under UNIX operating system. If the DOS window closes at once after users’ typing “startup”, it might be because users have not setup the java environment successfully. One possible reason is because we forgot to setup the environment variable “JAVA_HOME”. Now users can open a local web browser, type the address “http://localhost:8080/” or “http://computerName:8080/”. For instance, if a computer’s name is “CYPRUS”, in the web browser such as Internet Explore, user can type “http://CYPRUS.cs.concordia.ca:8080/” to start OntoXpl. As the result, the main menu page of OntoXpl shown in Figure 3.1 will appear on the screen, and ready to help the user explore the ontology easily and conveniently.

A.2 Appendix 2 OntoXpl API and package information

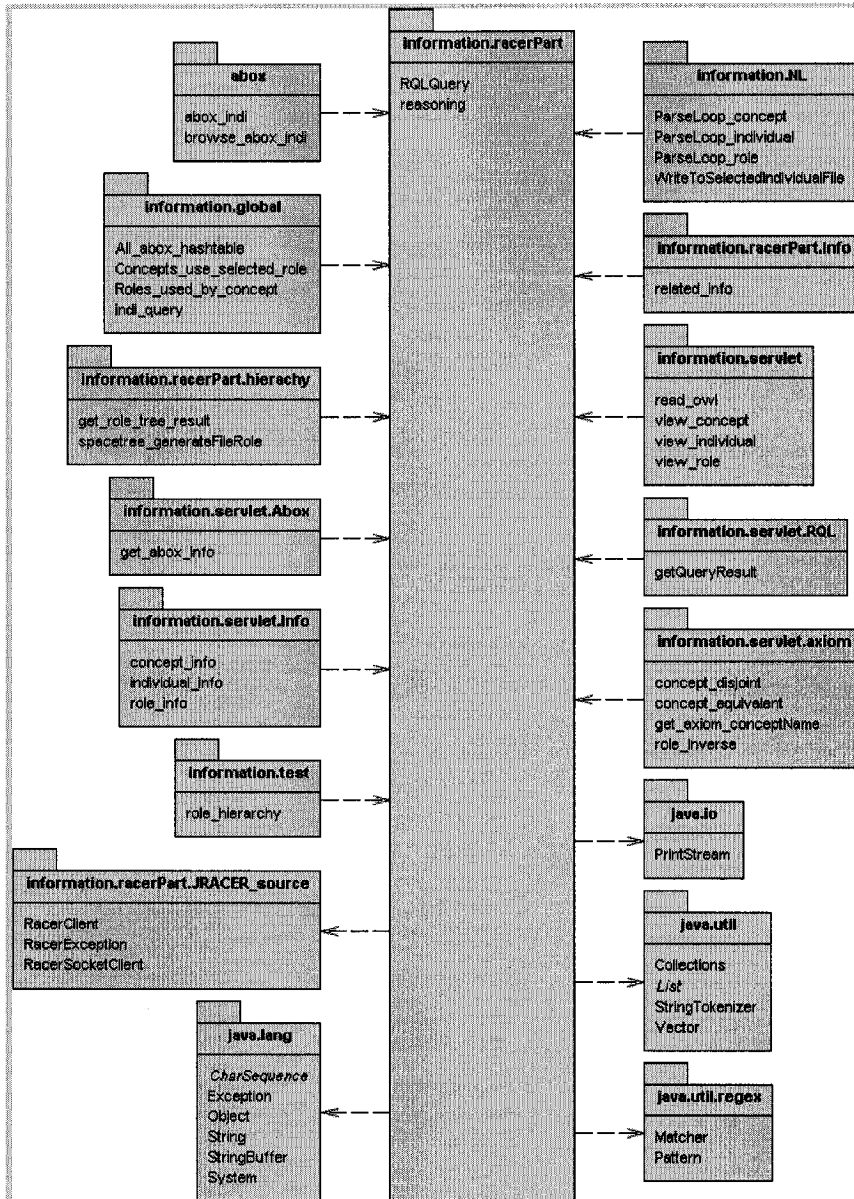


Figure A2.1: RacerPart Package Info

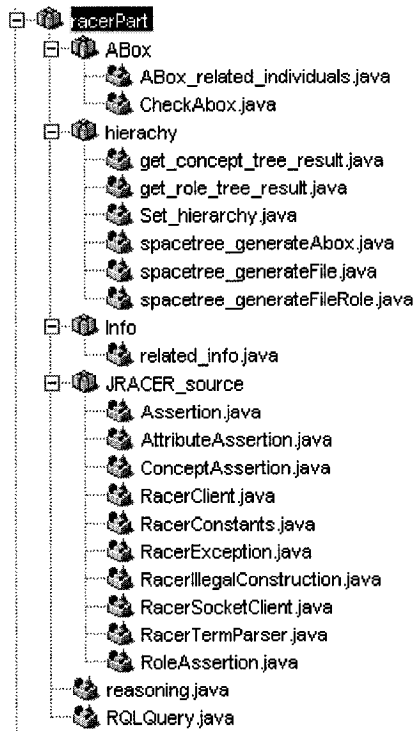


Figure A2.2: Reasoning racerPart

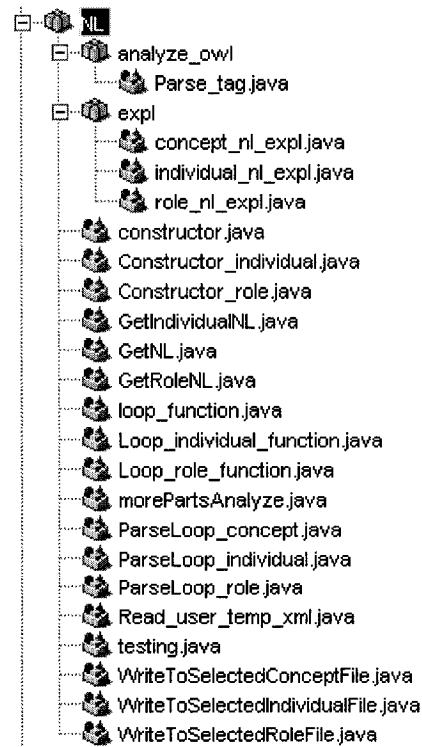


Figure A2.3: Natural language functions

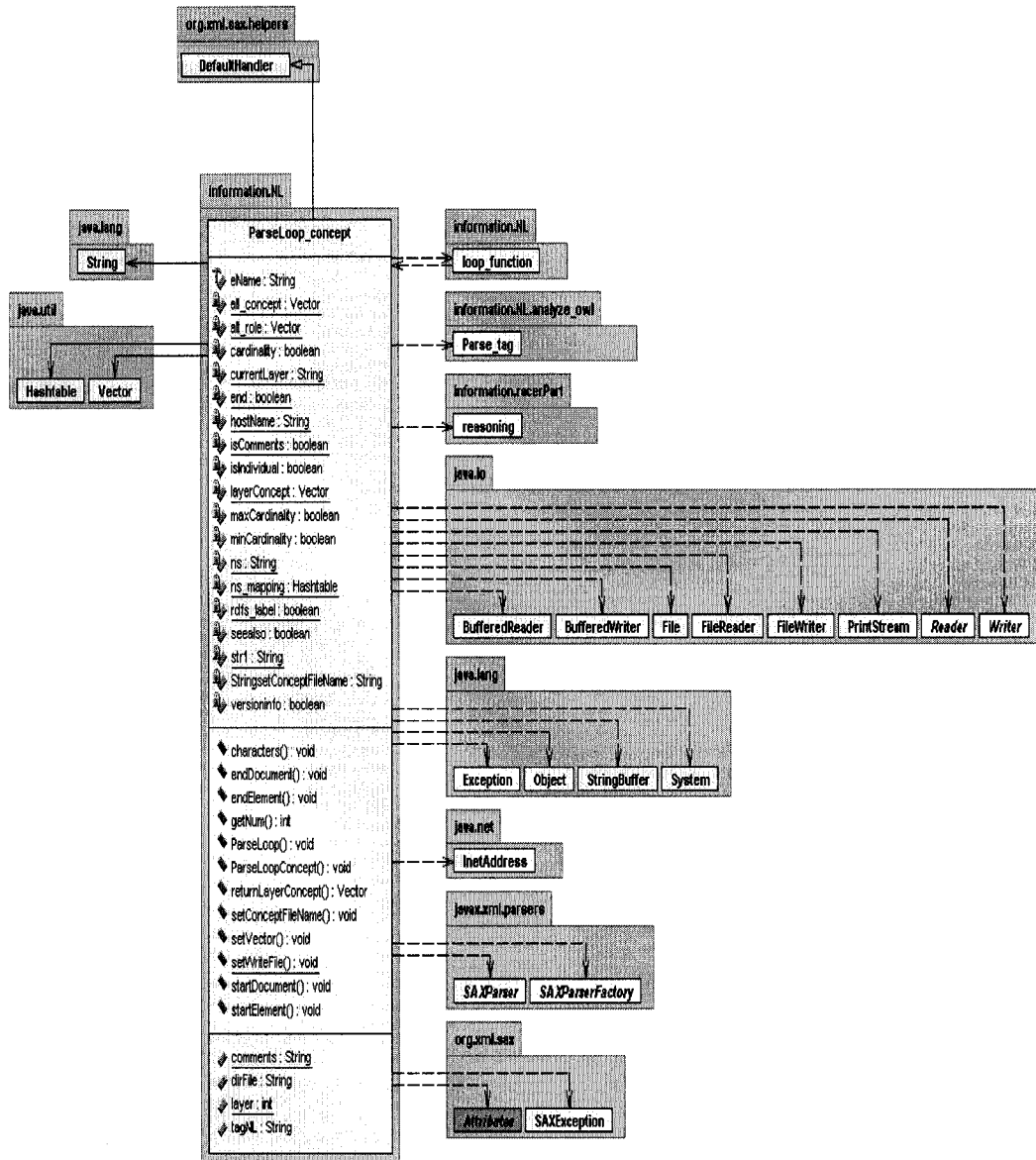


Figure A2.4: NL concept parsing class

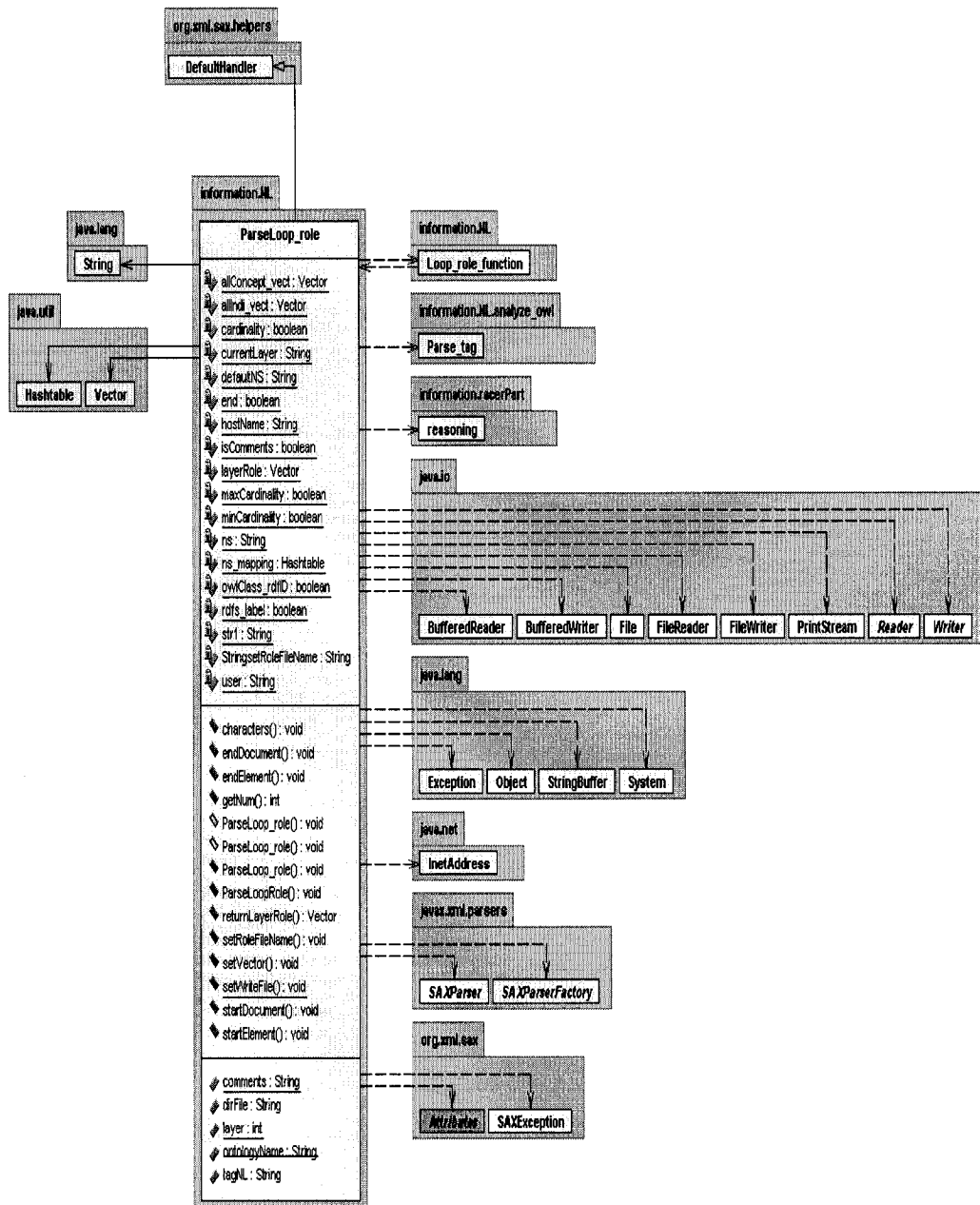


Figure A2.5: NL Role parsing class

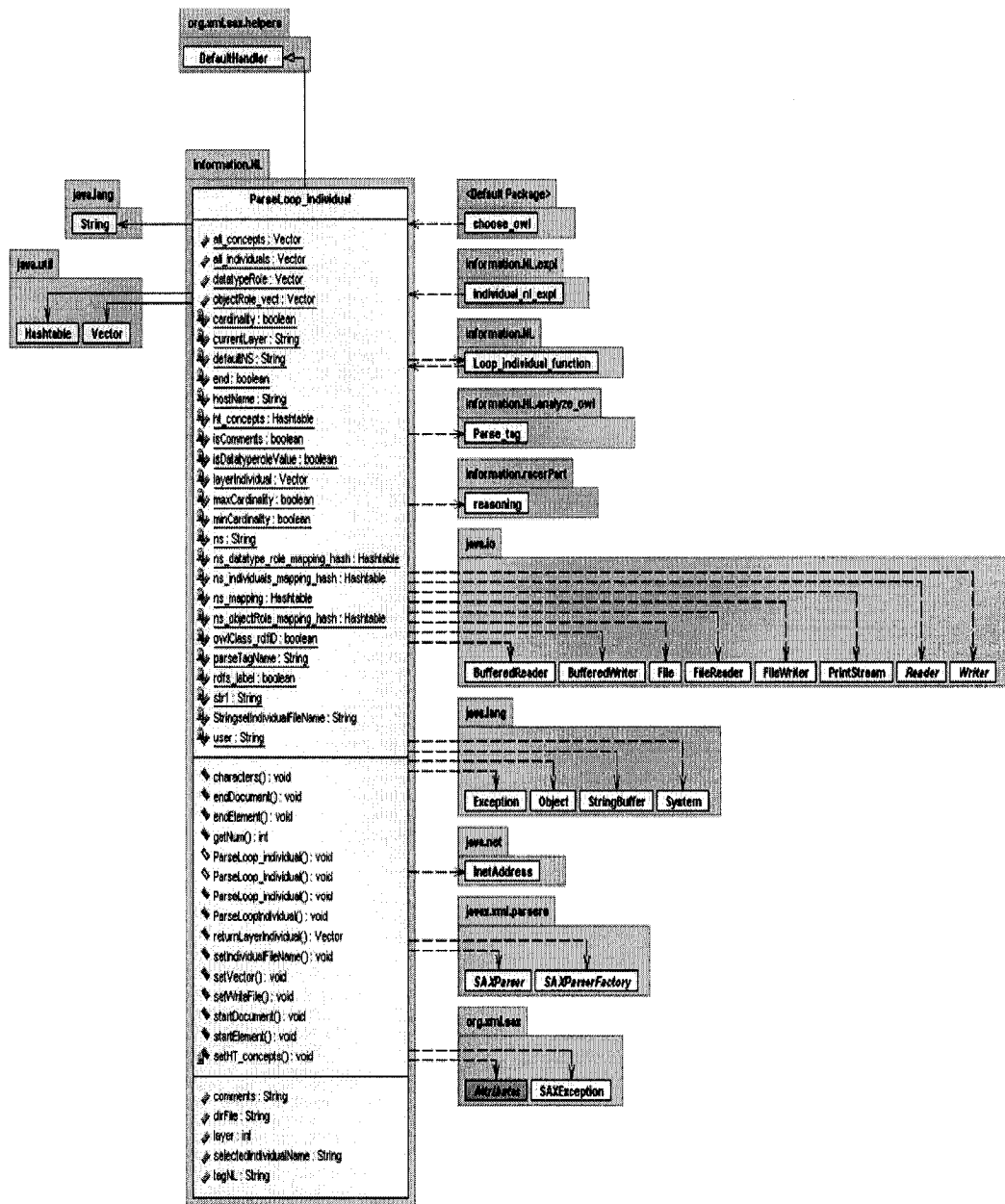


Figure A2.6: NL instance parsing Class

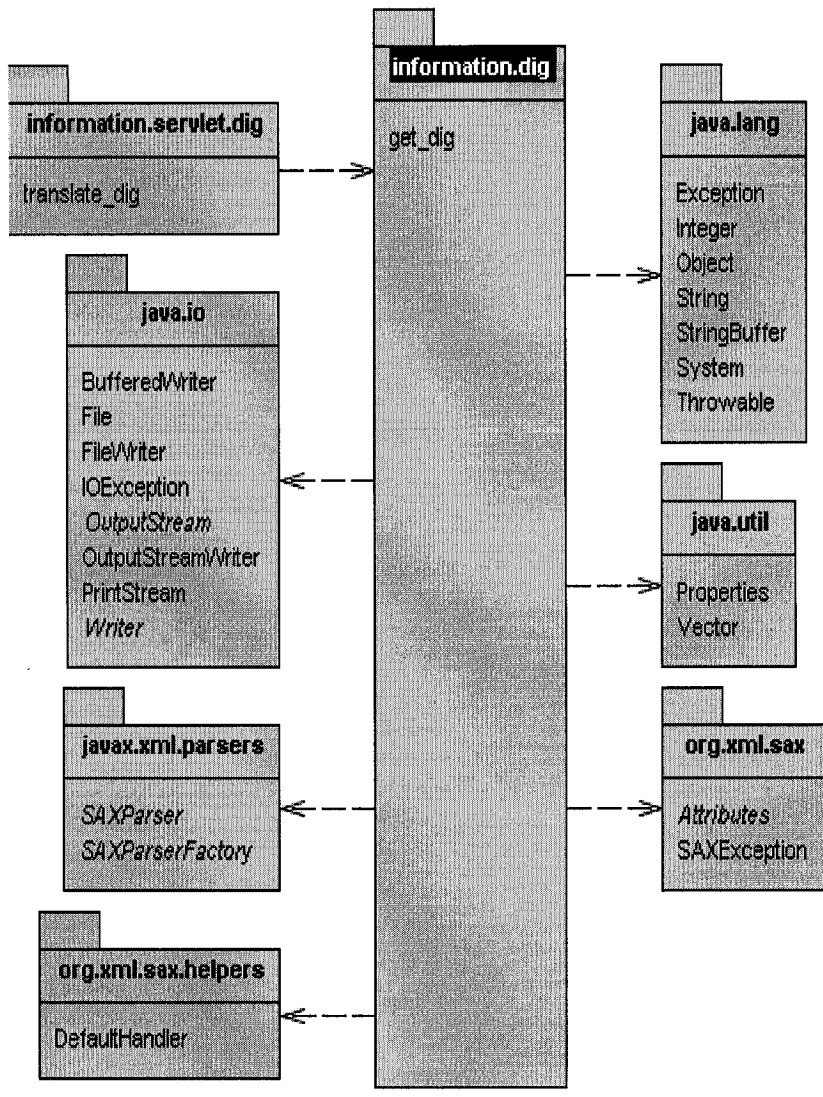


Figure A2.7: DIG translation function

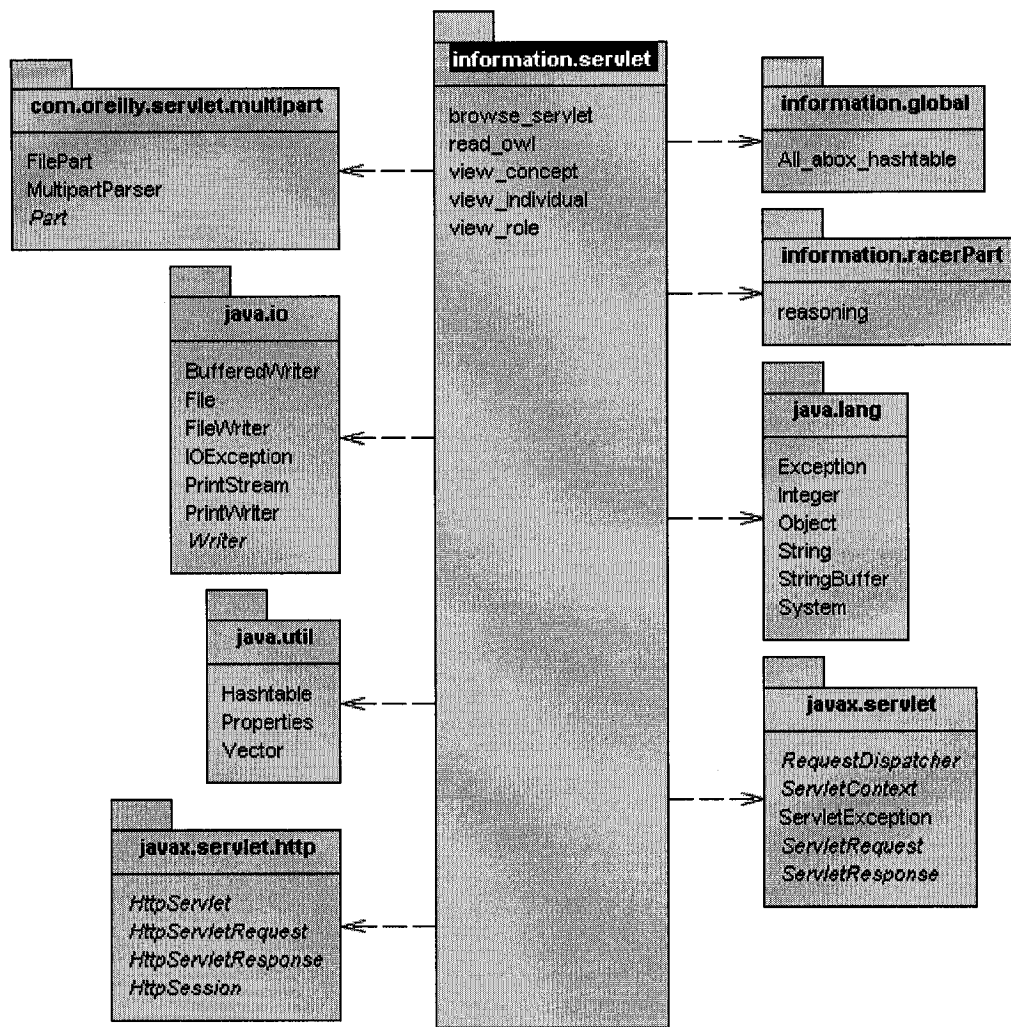


Figure A2.8: Servlet package information

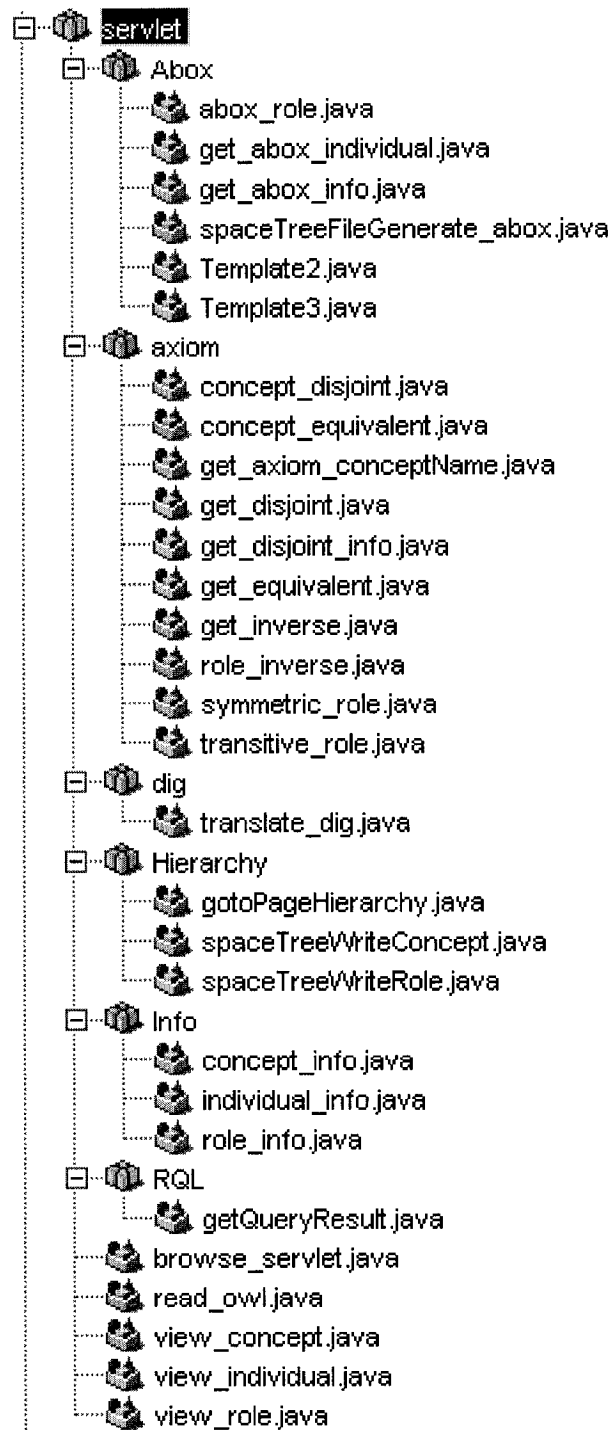


Figure A2.9: Servlet Structure