OntoVQL: Ontology Visual Query Language

Amineh Fadhil

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fullfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

September, 2008

Your file Votre référence
ISBN: 978-0-494-45461-9
Our file Notre référence
ISBN: 978-0-494-45461-9

ABSTRACT

OntoVQL: Ontology Visual Query Language

Amineh Fadhil

Applying visual techniques to access data has been a successful way of abstracting and simplifying the complexity of the operation to a user not versed in the technical domain. Visual querying systems have been developed to represent a database schema and to visually query its content. Similar effort has been done for visually representing the knowledge in an ontology. Since ontologies are gaining ground into playing an important role in representing knowledge of multidisciplinary domains, there grows the need for a simple way to query ontologies.

This thesis presents OntoVQL, a formal visual query language that allows to query OWL ontologies. OntoVQL hides the complexity of an OWL query language by abstracting the query as a graph that can be broken and reassembled and is designed to formulate more than simple queries with the and/or constructors. OntoVQI, a visual querying interface is introduced as the prototype that has been implemented to effectively formulate queries in OntoVQL and to view their results. This prototype facilitates query formulation by providing features such as query preview.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 General motivation for database visual querying

It has been shown that applying visual techniques for accessing data is particularly successful [10]. This is because of the increase of non-expert and casual users who are not familiar with database querying languages such as SQL. Moreover, in comparison to the use of natural language, visual querying overcomes the problems of depending on the user's language and the limitation imposed by the application area [10].

1.2 What is a visual query system?

Visual query systems (VQSs) are defined as systems for querying databases that use a visual representation to depict the domain of interest and express related requests [9]. These systems are based on the advantage in the ease of grasping knowledge visually; allowing the recognition and handling of large quantities of information [9]. Moreover, the human-computer dialogue is improved by the possibility of using visual feedback [9]. A visual query system provides a user-friendly visual interface for accessing a database [10]. It is composed of a visual query language (VQL) for pictorially expressing queries and of some functionality for facilitating the human-computer interaction [10]. Target users of a VQS lack technical skills and usually ignore the inner structure of the database they are accessing. Many VQSs employing various visual representations and interaction strategies have been proposed in the literature. A survey on visual query systems for databases can be found in [9].

1.3 What is a visual query language?

Visual query languages are issued from visual programming languages that were first introduced in [10]. In fact, visual programming languages express abstract and non-abstract objects in a visual representation in order to enhance man-machine interaction. This class of visual programming languages subclasses the visual query languages that deals with data in databases. There exist iconic and graphical VQLs. The former is based on icons and the latter based on diagrams. "Graphical languages are more suited to be formalized, given the precise mathematical structure (i.e., the graph) the diagrams are based on" [10]. The formalization of the language permits one to evaluate its expressive power.

1.4 Research goal and outline

As it will be seen in the literature review of Chapter 2, a number of Visual Query Systems have been developed in the context of databases. However, even though some work has been done for querying OWL ontologies, it is not comparable to the database domain given mostly that the use of ontologies is recent. Most of the attempts of visually querying an ontology was done in context of offering an integrated view of multiple databases. Very little has been done for effectively visually querying an OWL ontology. In that context, the main objective of this research is to develop a Visual Query Language for querying ontologies. The goal is to design a VQL that is formal, simple to understand and allows to formulate more than simple basic queries. The idea is to design a VQL that abstracts the complexity of the underlying logical querying language. It should be possible to map the visual constructs in the VQL to the semantics of any querying language for ontologies.

In our case, we present OntoVQL, the designed Visual Query Language that is mapped to a subset of the semantics of nRQL. In the next chapter, an overview of Description Logics, OWL ontologies and nRQL is given to lay the background of this research. The major work for visually querying databases and ontologies is covered in Chapter 2 to explain how they helped in the design decisions of our VQL. The Visual Query Language is then presented in the following Sections (3.3, 3.4, 3.5) that cover its visual notations, syntax and mapping to nRQL. Finally, OntoVQI is presented in Section 3.6 as the prototype that has been implemented for querying ontologies with OntoVQL.

Chapter 2

Overview of visual query systems

In this chapter, we will make a brief overview of existing visual query systems for databases as well as some visual querying interfaces for ontologies. These will be covered as a background to the visual system we will be introducing in Chapter 3.

2.1 Related work in database visual querying

2.1.1 Hyperflow

Hyperflow [14] is a visual language that merges visual query languages (VQLs) and visual dataflow languages features. It is designed for the purpose of facilitating information retrieval from databases. In fact, it allows to visually construct and execute information analysis processes in a single diagram. Hyperflow has a twofold motivation. First, up to now many databases and analysis services offer only basic, under-expressive user interfaces that do not allow to perform operations such as arbitrary joins, grouping or aggregate functions which are usually performed on databases. Therefore, Hyperflow supports almost all constructs from the SQL, OQL and RDF query languages. Thus, it gives the user a unified language to most common databases. Second, the issue of data integration: databases and analysis services are highly distributed and different formats for the same data are used in different databases. On that account, in order to ease the process of transforming complex database queries into information handling processes, Hyperflow offers the possibility of combining queries into workflows and provides an expressive and simple way in making a query and specifying a workflow. Two modes of operation exist in the Hyperflow language. The first one consists of allowing the user to design and execute the query and the second one is for interactively exploring information. As an example to illustrate how Hyperflow can be employed, consider the following simple bioinformatics analysis where a researcher wishes to analyse a DNA molecule by comparing its sequence to sequences in public databanks depicted in Figure 2.1. The Hyperflow language introduces a mix of declarative graph-based constructs and functional dataflow constructs. The graph-based constructs permit to determine the subgraph of the ontology the user is interested in, whereas the dataflow constructs are meant to represent the remaining parts of the query such as Group-By or Order-by. An easy-to-use user interface is provided for the user. Queries usually start by simply dragging a class icon from the ontology browser into the query box. Then, context-sensitive popup-menus guide the user into building up the query. Another useful ability is that sections of the workflow can be reused for constructing other queries.

In Figure 2.1, (a) shows the execution of a BLAST service that finds alignments to similar sequences in the databank. Then, in (b) a follow-up query is posed by using the obtained alignments from (a) to get mammalian sequences whose similarity of alignment is more than 98%. In (c), a global alignment algorithm, one for each pair of the original and retrieved sequences is conducted. In (d), the figure shows a service that gives a multiple alignment of the retrieved sequences. Finally, in (e) a workflow is prepared that finds which transcription factors are common to all the sequences in the group. [14]

2.1.2 GLASS

Glass [31] (Graphical Query Language for Semi-Structured Data) is a graphical query language for users to extract information from semi-structured data. It was developed

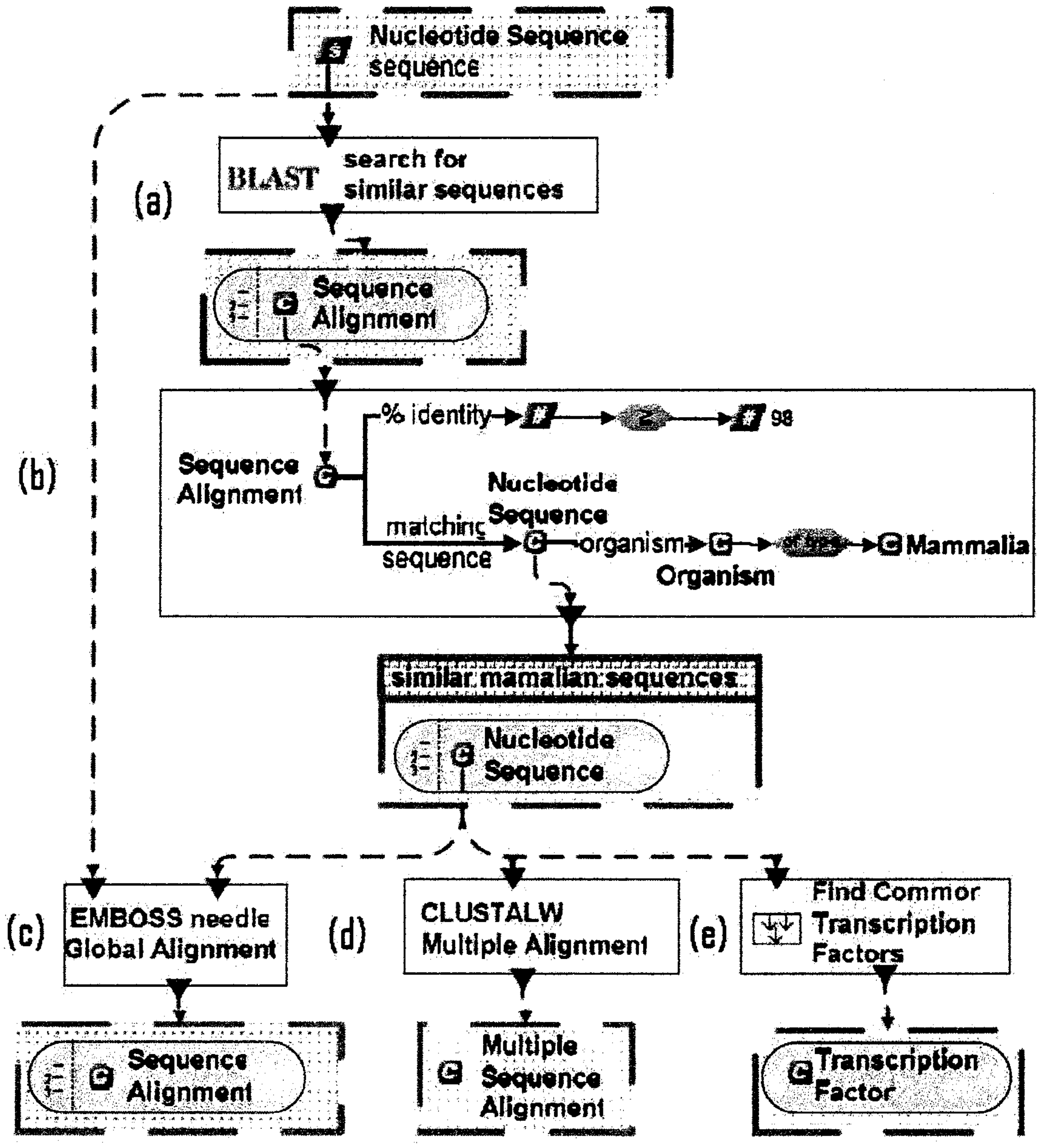


Figure 2.1: Hyperflow flow of execution. (Figure taken from [14])

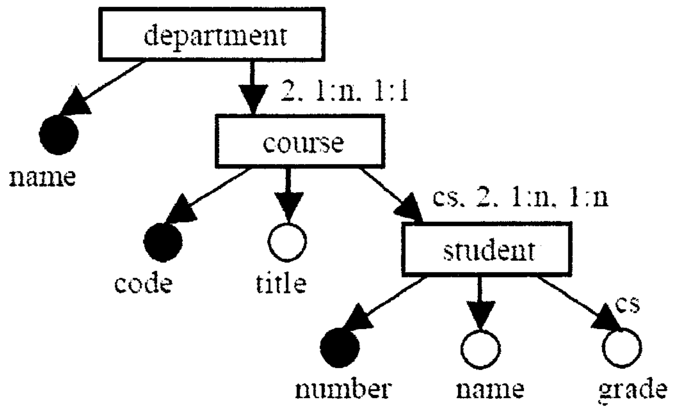


Figure 2.2: An ORA-SS schema diagram. Object classes are represented as squares. Attributes under the corresponding object classes are represented as circles. Arrows indicate the nested structure of the schema and the labels on the edges are for relationship types. (Figure taken from [31])

in order to exploit the full power of XML documents as well for providing an easy-to-use interface since XQuery, the current standard for querying XML data, is too difficult for common users to use. Most visual querying languages are based on the visualization of the data model. The data model used in GLASS is ORA-SS (Object-Relationship-Attribute model for Semi-Structured data). Figure 2.2 gives an idea of what an ORA-SS schema diagram looks like.

The basic query operators (Selection, Projection, Join, Aggregation, etc.) that are visually represented are related to the basic ones usually found in database query languages. Figure 2.3 depicts an example of a simple selection query based on the above schema diagram. The query is separated in two parts; the LHS is used for specifying the conditions whereas the RHS is used to define the output. Moreover, existential and universal quantification as well as negation are expressed textually within a logical condition window.

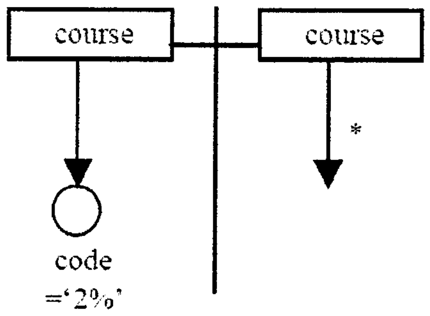


Figure 2.3: Selection query. This query selects all courses whose codes begin with "2". The main drawback of GLASS is that it combines graphs with text. Therefore, on top of requiring the user to learn the basic concepts and how to use them to build a query, the user must know how to write logical expressions. (Figure taken from [31])

2.1.3 Klaeidoquery

Klaeidoquery [29] is a visual query language with the same expressive power as OQL (Object Query Language). The motivation behind this work relies on the problems encountered with textual queries. In fact, the database user needs to know the database classes, attributes and relationships structure before writing a query. Also, there are the problems related to the syntactic and semantic errors. The query is depicted as a filter-flow since the objective is to show the refinement of information through the query. Therefore, the query follows the filter-flow model. Thus, the input to the query consists of instances (extent of the database) that flow through the query and get filtered according to specified constraints. The output can then be further refined by being an input for another query. As it is the case with most graph-based visual languages, a query formulation relies upon the selection of parts of the view of the database schema, hence, the importance of the visualization of the database schema. For this purpose, Klaeidoquery adopted icons with a textual description (Figure 2.4) under the claim that this representation gives a better comprehension than icons alone or text alone [6, 20].

A query is composed of three main parts: the extent of the database as the

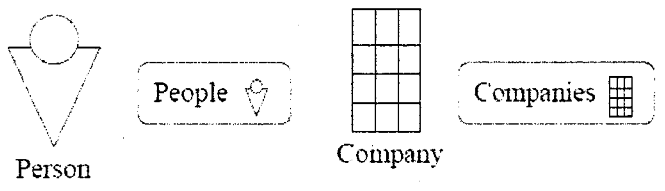


Figure 2.4: Example Schema for Klaeidoquery. Class icon and the class name along with the extent name and the icon associated with the class of the extent. (Figure taken from [29])

initial input to the query, classes from the database schema and query constructs for specifying constraints. A simple select query is visualized along with its corresponding SQL equivalence in Figure 2.5(a). Constraints to restrict the query result can be added so that the instances get filtered through each constraint that could be and-ed or or-ed with other constraints, or even negated (Figure 2.5(b)). Furthermore, universal and existential quantifications can also be visually depicted (Figure 2.5(c)).

2.1.4 Gql

Gql [33] is a declarative graphical query language based on the functional data model (i.e. the functional view of the binary relational model). It is a declarative language since the query describes what is the desired result rather than the procedure of how to obtain the result. The authors of Gql have designed the language for users working with databases but without necessarily being versed in programming. They claim that it should facilitate and hasten the database programmer job. Gql's expressivity is a superset of that of SQL. Also, even though the following might be debatable in the sense that it sounds more like a subjective opinion, but according to the authors, the design principles on which Gql was based on are: ease-of-use, expressivity and ease-of-use must scale up, the query should be fully expressed in a single diagram, it should be formal and finally it should be separate from the user interaction procedures. The reason why diagrams were adopted for representing queries is for their superior

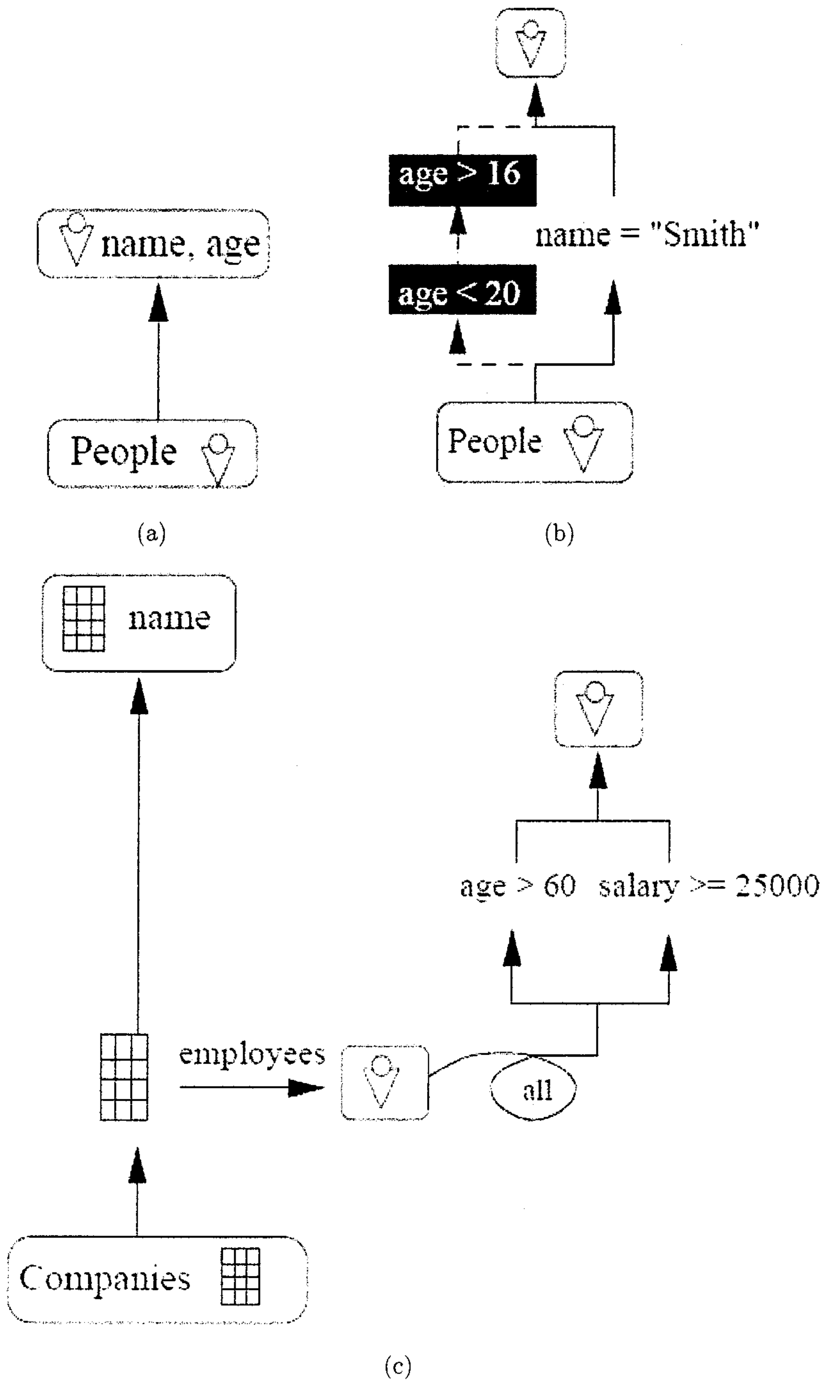


Figure 2.5: (a) Selecting attributes. Certain attributes of the class Person are selected for output. (b) Visualizing "and", "or" and "not". Constraints on the same line are AND-ed. Constraints on separate branches are OR-ed. Constraints that are selected are negated. (c) Universal quantification. This query uses the "for all" operator of OQL. The "exists" operator can be used instead for existential quantification. (Figures taken from [29])

expressivity over textual representations [26]. Moreover, the functional data model has a diagrammatic representation (Figure 2.6).

A query over a schema would consists of parts of that schema put together with some query constructs to allow query specification (Figure 2.7). The box construct turns out to be the most important query construct in Gql. In fact, every query is enclosed by a box (the outer or top-level box) and nesting of boxes allows nesting of results, aggregates, negation, disjunction, universal and existential quantification. Figure 2.8 shows how negation, disjunction and universal quantification are visualized with different types of box constructs. However, different conditions are AND-ed by default as it is shown in Figure 2.7.

2.1.5 GraphLog

GraphLog [12] is a visual query language that evolved from G+ and is based on the graph representation of the data model. In addition, it has the same expressive power as first order logic with transitive closure or stratified linear Datalog. In fact, GraphLog is formally defined by graphical set theory. Thus, the database schema is represented within a directed labelled multigraph G (Figure 2.9) that is formally defined.

Like it is the case with other visual query languages, GraphLog profits from the graph representation of databases to express queries. As a matter of fact, queries are graph patterns that are searched in the database graph. Each pattern defines a set of new edges (i.e. new relations) by mean of distinguished edges that are added to the graph whenever the pattern is found. That is, whenever the edges of the graph pattern are present, then the relation defined by the distinguished edge holds. A query pattern is formally defined as a directed labelled multigraph with a distinguished edge. An example of a GraphLog query is given in Figure 2.10(a) that intuitively expresses the predicate no-desc-of(P1,P2,P3) with the descendants P3 of person P1 who are not descendants of person P2. The graph pattern is translated into a logic program as

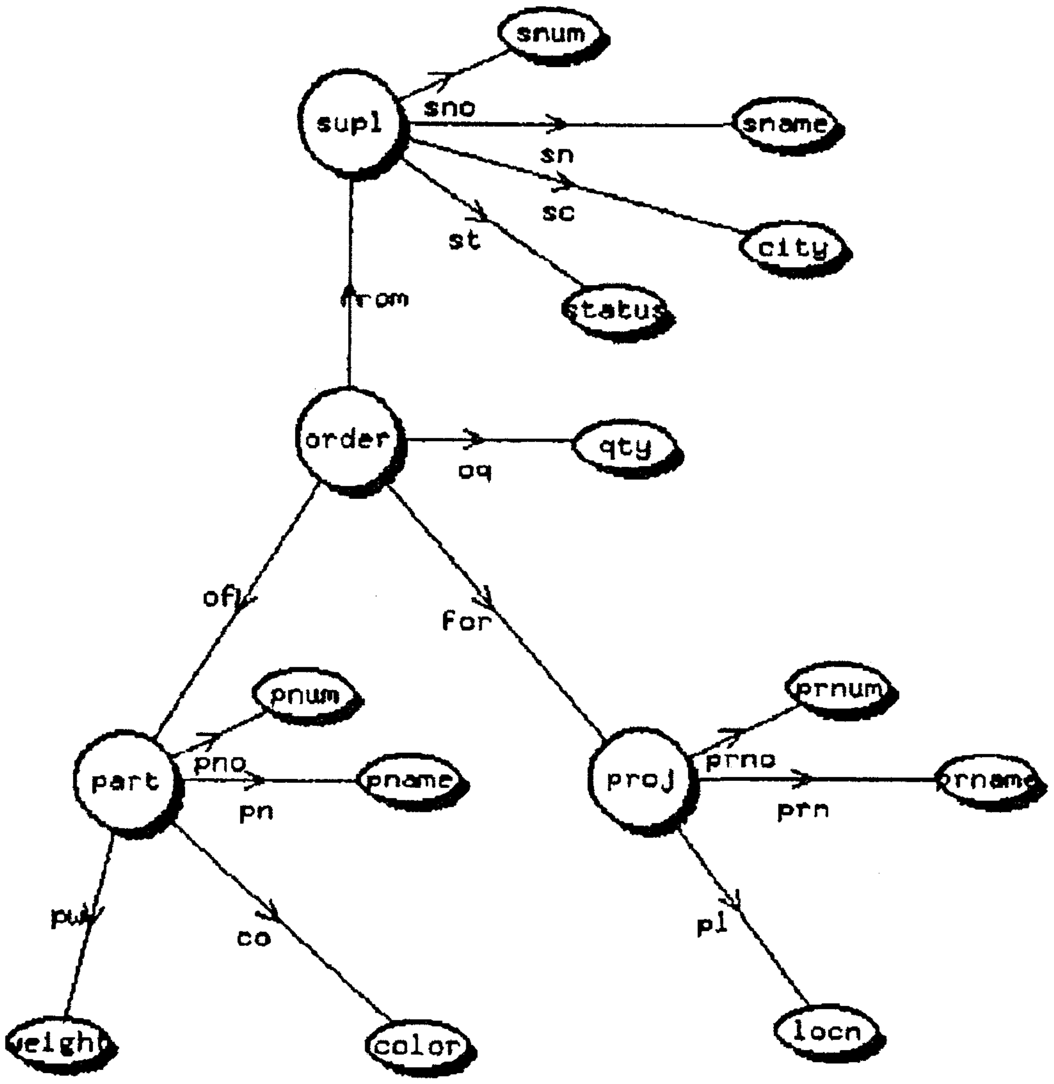


Figure 2.6: Entities are represented as nodes; circles are real world entities whereas ovals are lexical entities. The functions between entities are represented as labelled directed arcs between nodes. (Figure taken from [33])

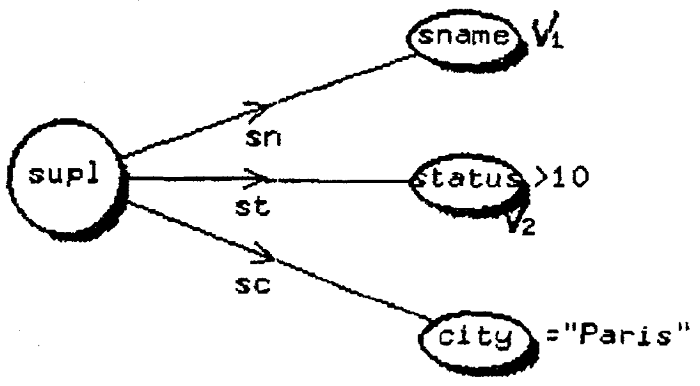


Figure 2.7: A sample query over the schema of Figure 2.6. (Figure taken from [33])

stated in Figure 2.10(b).

2.1.6 GOQL

GOQL [22] is a visual query language that provides a user-friendly graphical interface to support ad-hoc queries for object-oriented database applications. This work was motivated by the fact that most graphical query languages are easier to use and learn than textual languages [33]. This is because users need not to be aware of a particular syntax or remember built-in words and constructs. Thus, rather than writing obscure code, users can visualize and represent their queries diagrammatically or graphically. GOQL can express any query that can be expressed in OQL which is the standard SQL-like object-oriented query language of the O2 object-oriented database system. In order to hide from the user the underlying object-oriented features of that database, the formulation of a query is based on a graphical query model that is proposed for the database schema. An example of such a graphical schema is shown in Figure 2.11.

There is no formal definition for the graphical representation of the queries. Some query constructs such as selection, conjunction, disjunction and universal quantification are illustrated in Figure 2.12 and Figure 2.13 along with their corresponding OQL query.

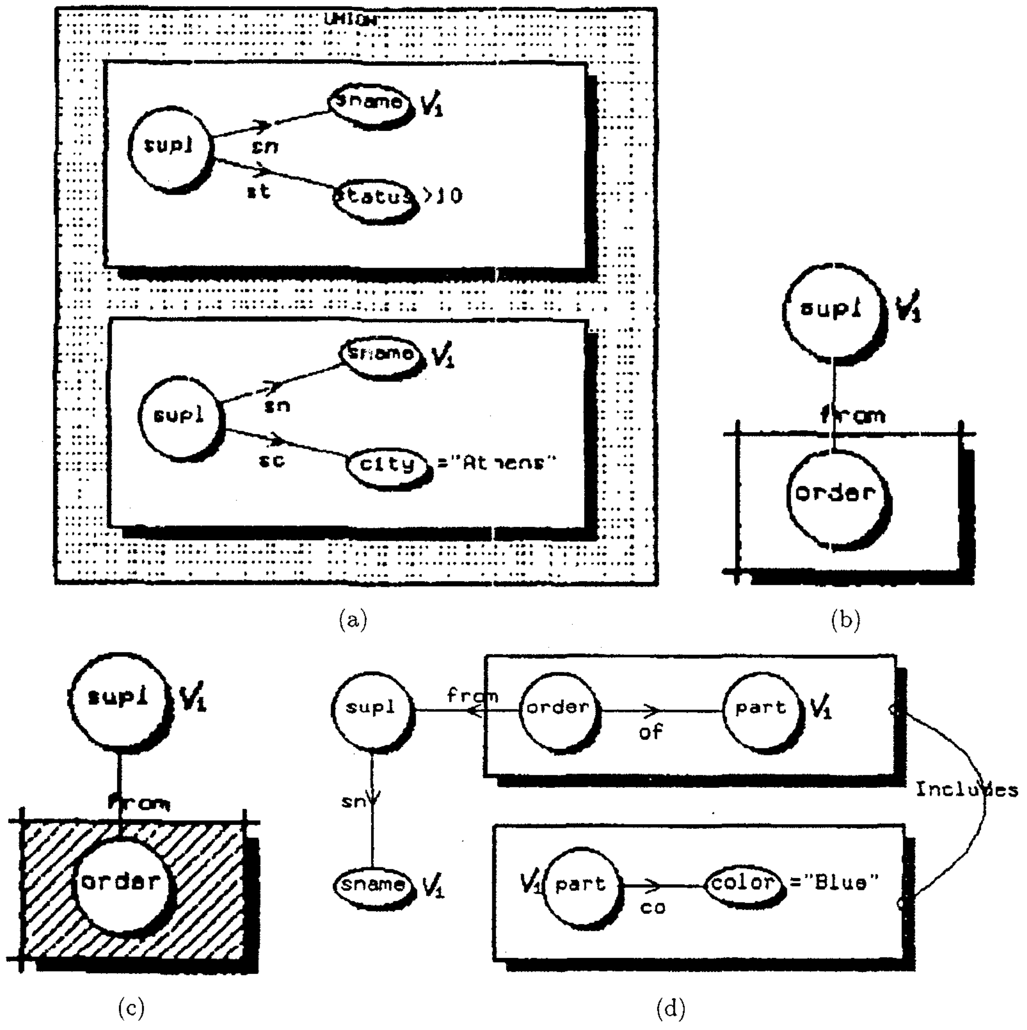


Figure 2.8: (a) Union is represented with the union collection box (UCB) constructs that must contain two or more union compatible collection boxes for representing the union of these collections. (b) Negation is represented with the truth value box (TVB) constructs. The first query shows a positive TVB corresponding to an existential quantifier whereas the second query shows the negative TVB corresponding to the negated existential quantifier. (c) Universal quantification is represented with collection comparison operators (CCO) which are a new labelled edge connecting two union compatible collection boxes. The depicted query corresponds to the following textual meaning: "get every supplier who has an order for every blue part". (Figures taken from [33])

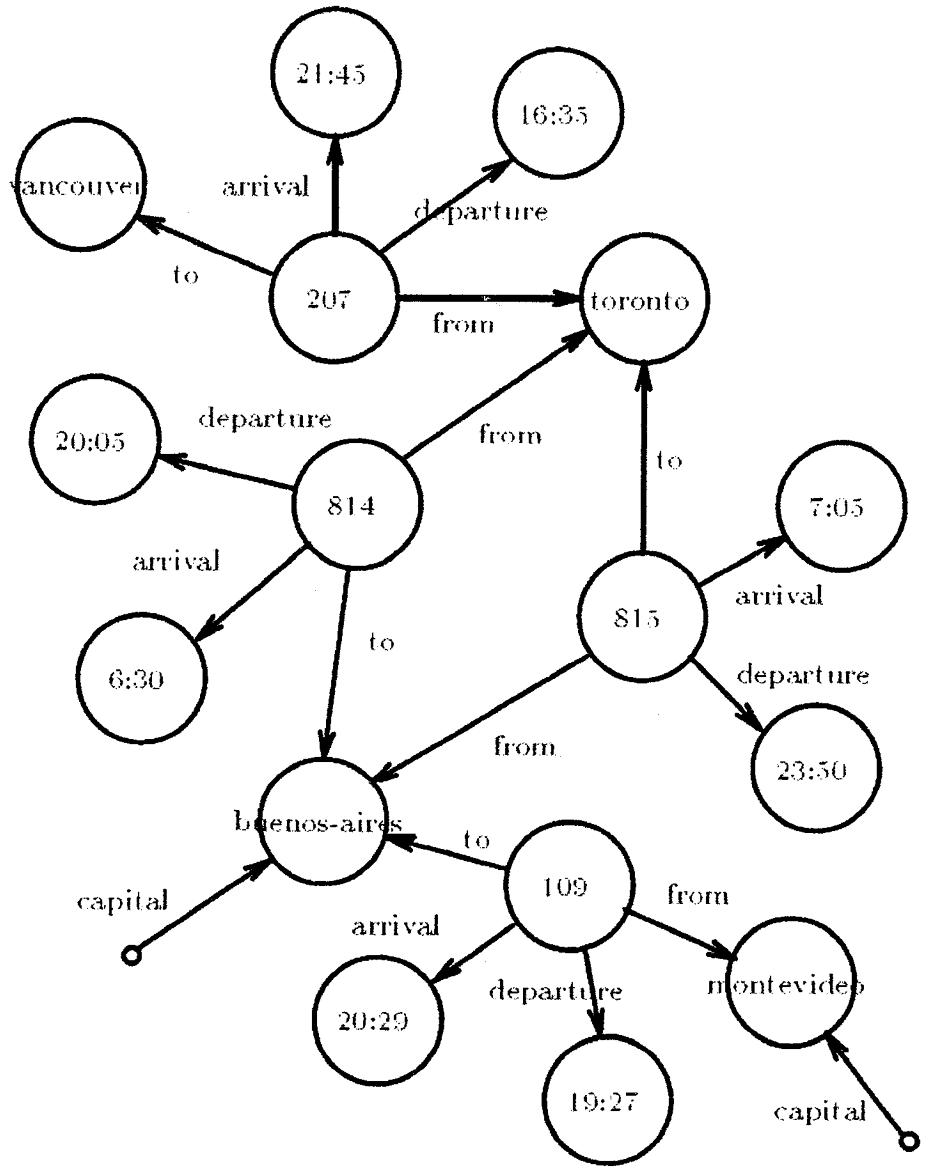
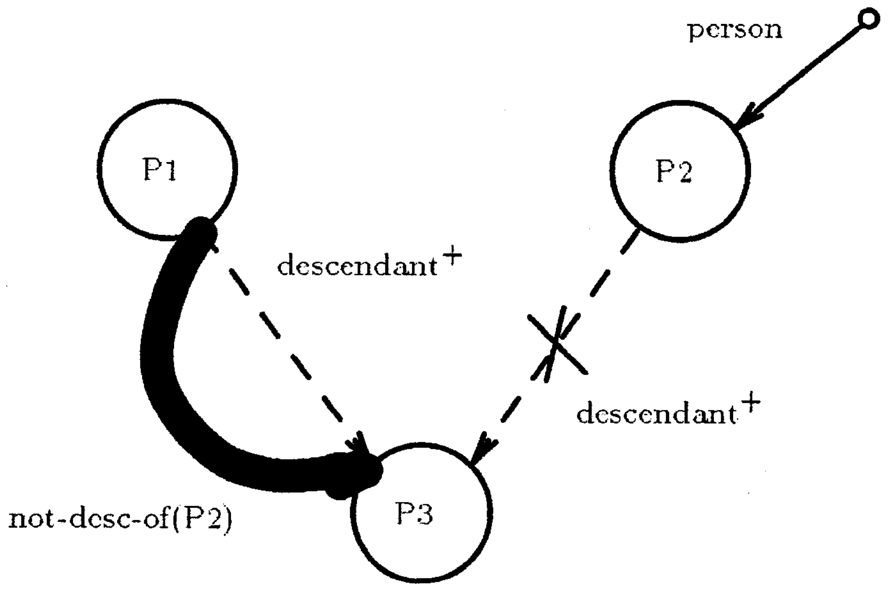


Figure 2.9: A graph representation of a flights schedule database. (Figure taken from [12])



(a)

```
not-desc-of(P1,P3,P2) ← descendant-tc(P1,P3),
                         ¬ descendant-tc(P2,P3),
                         person(P2).

descendant-tc(X,Y) ← descendant(X,Y).
descendant-tc(X,Y) ← descendant(X,Z), descendant-tc(Z,Y)
```

(b)

Figure 2.10: (a) The descendants of P1 which are not descendants of P2. (b) The descendants of P1 which are not descendants of P2 in Datalog. (Figures taken from [12])

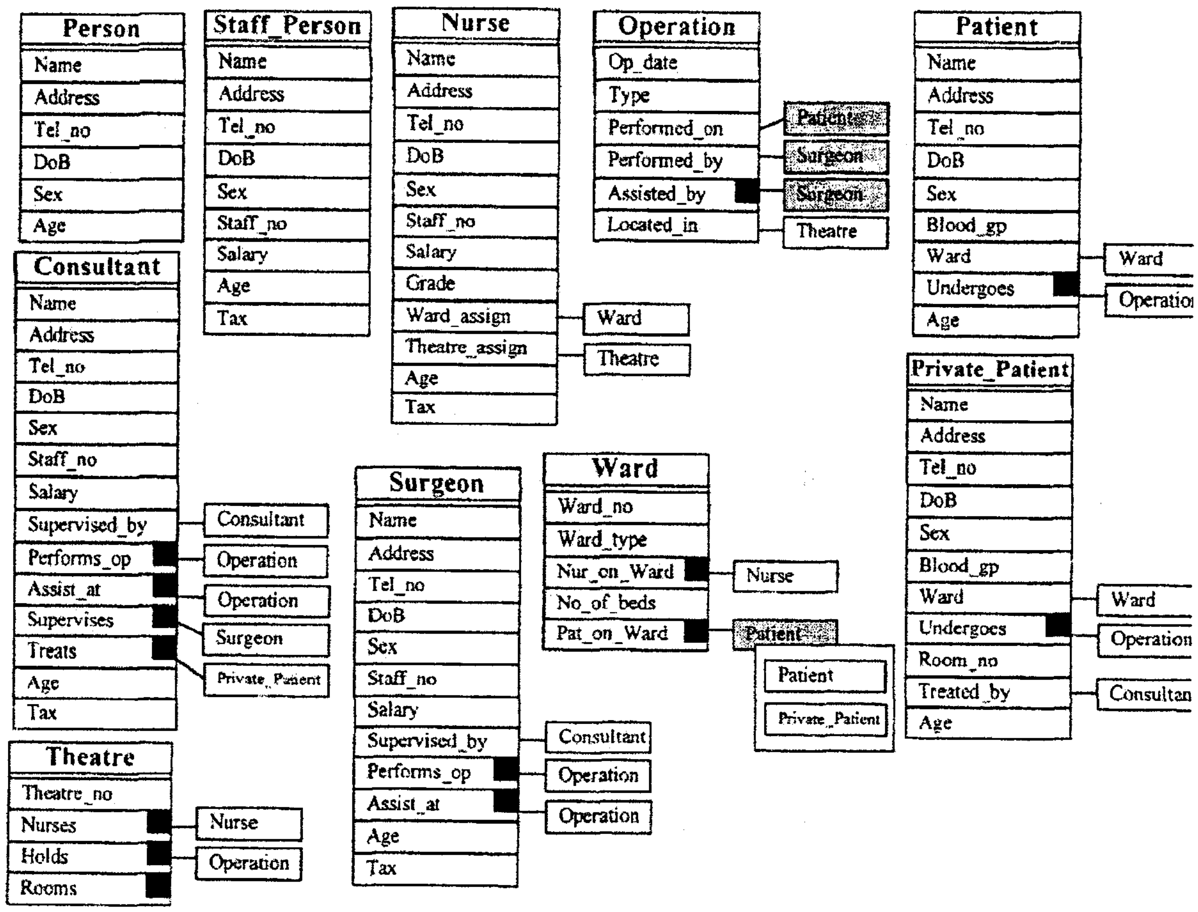
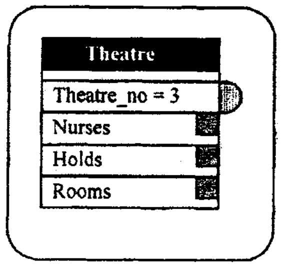


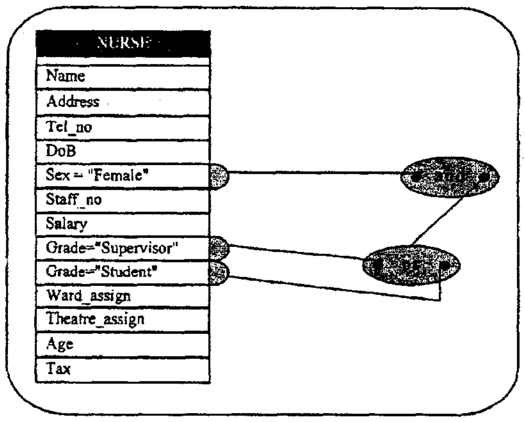
Figure 2.11: Example of graphical database schema in GOQL. (Figure taken from [22])

```
select t
from t in Theatre
where t.Theatre_no = 3
```



(a)

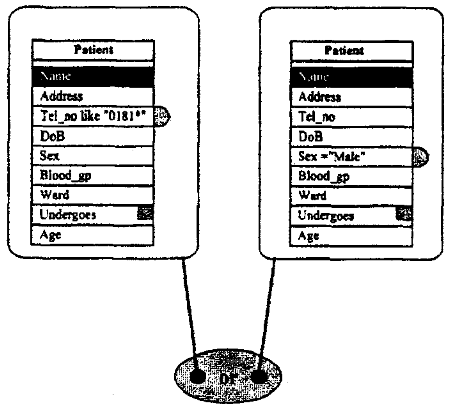
```
select t
from t in Nurse
where (t.Grade = "Student" or
        t.Grade = Supervisor")
   and t.Sex = "Female"
```



(b)

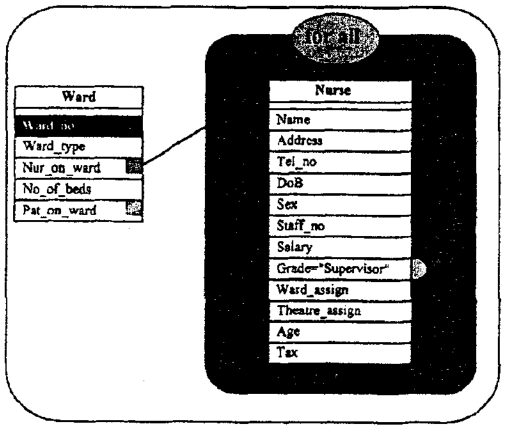
Figure 2.12: Example queries of GOQL (Figures taken from [22])

(**select** p.Name
from p **in** Patient
where p.Tel_no like "0181*")
union
(**select** p.Name
from p **in** Patient
where p.Sex = "Male")



(a)

select w.Ward_no
from w **in** Ward
where for all wn in w.Nur_on_ward:
  wn.Grade = "Supervisor"



(b)

Figure 2.13: Other example queries of GOQL (Figures taken from [22])

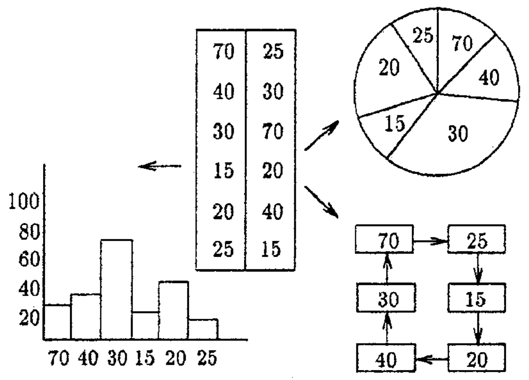


Figure 2.14: Visualizations of a binary relation. (Figure taken from [13])

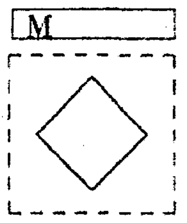
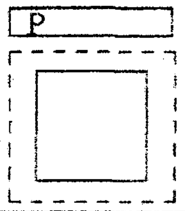
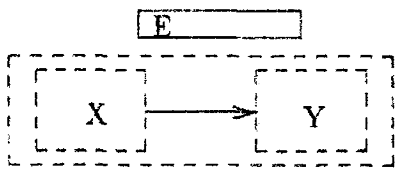
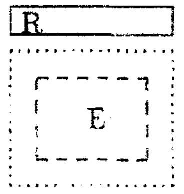
2.1.7 DOODLE

DOODLE [13] is a visual and declarative language for object-oriented databases and its semantics is given by F-logic [23]. The main feature behind DOODLE is that it allows to display and query databases with arbitrary pictures. In fact the user must specify how the data is to be visualized. However, it is possible to transform the display from one visualization to another (Figure 2.14). Likewise, the user can specify the display for querying as well. Nevertheless, the visual query language should ideally be close to the visualization of the database. This allows to perceive the queries as patterns to be matched against the database.

Data and query visualization are defined by a set of conventions for obtaining pictures from data. In fact, the specification for the visualization has to be done through a visual program. The semantics to the visual program is given by showing how they can be translated to F-logic programs (Figure 2.15).

2.2 Related work in ontology visual querying

The work done in the ontology visual querying field is mostly about providing an integrated logical view of heterogeneous databases by offering the data sources view and querying to be done at the ontology level [24]. This effort was conducted in the context of Large Data Integration projects like TAMBIS [36], KIND [27], SEEK [28],

softGraph	f-language
	M:*module*
	P:*procedure*
	E : *contains* [outer → X : *block*, inner → Y : *block*]
	R : *agg* [members → {E}]

(a)

```
M[display@softGraph → vis(M) : diamond, defbox@softGraph → vis(M)] ⟵ M : module.
P[display@softGraph → vis(P) : square; defbox@softGraph → vis(P)] ⟵ P : procedure.
E[display@softGraph → vis(E) : arrow[from → U, to → V]; defbox@softGraph → vis(E)]
                         ⟵ E : contains[outer → X, inner → Y],
                            X[defbox@softGraph → U : visualObject],
                            Y[defbox@softGraph → V : visualObject].
R[display@softGraph → vis(R) : visAgg[visMembers → {X}]]
                         ⟵ R : agg[members → {E}], E : contains[defbox@softGraph → X : visualObject
```

(b)

Figure 2.15: a) Visual program that defines softGraph. (b) F-logic program for softGraph. (Figure taken from [13])

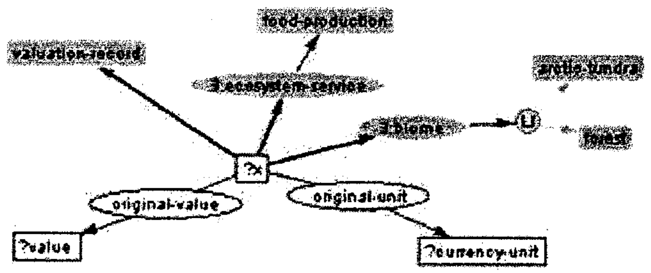
and SEWASIE [8].

2.2.1 SEWASIE

SEWASIE is presented as an intelligent user interface whose goal is to provide the user with support in formulating a precise query even when completely ignoring the vocabulary of the underlying information system holding the data. The visual query is then translated to a Select-Project-Join SQL query that is executed by an evaluation engine associated to the information system. Users start a query by choosing one of the pre-prepared domain-specific patterns presented to them. Afterwards, they can refine the query by extending and customizing it. The refinement to the query consists of additional property constraints to the classes or a replacement of a class by another compatible class such as a subclass or superclass.

2.2.2 GrOWL-Query

GrOWL [25] is a tool that offers a graph based interface with graphical icons to depict different types of nodes (class, property, or individual) as well as language constructs (negation, union, etc.). The GrOWL visualization model is an accurate mapping of the underlying Descriptions Logics (DL) semantics of OWL ontologies. A conjunctive query consists of the conjunction of query atoms x:C and (x,y):R, where x and y are variables or individuals, C is a concept expression and R is a role. By introducing two types of variables: "select" variables prefixed by "?", and "ignore" variables prefixed by "-", it is possible to formulate a graphical query since the set of select variables would be defined by the query condition. Therefore, a GrOWL-Query is GrOWL with variables. In other words, a query is a GrOWL ABox diagram where the variables are allowed in place of individuals. For example, the GrOWL-Query diagram in Figure 2.16 (a) is the visual counterpart of the query in (b).



(a)

$$
\begin{aligned}
&answer(?x, ?value, ?currency\text{-}unit) : - \\
&\quad ?x : valuation\text{-}record, \\
&\quad \wedge ?x : \exists ecosystem\text{-}service.food\text{-}production, \\
&\quad \wedge ?x : \exists biome.(forest \sqcup arctic\text{-}tundra), \\
&\quad \wedge (?x, ?currency\text{-}unit) : original\text{-}unit \\
&\quad \wedge (?x, ?value) : original\text{-}value.
\end{aligned}
$$

(b)

Figure 2.16: (a) GrOWL-Query diagram. (b) The equivalent translation in DL. (Figure taken from [25])

2.2.3 GRQL

In the context of Semantic Web Portals, there have been several browsing interfaces that were proposed, such as OntoWeb [7], and OntoPortal [21], for the purpose of accessing ontologies and their related information sources. These visual browsing interfaces offer a graphical view of the entire ontology as a tree or a graph of related classes and properties where users can either access directly the classified resources or formulate filtering queries. GRQL [2] relies on the full power of the RDFS [7] data model for constructing queries expressed in a declarative language such as RQL [21]. The user starts constructing his query by choosing an entry point and discovers the RDFS class and its property definitions. Then, he can continue browsing by generating at each step the RQL path expressions required to access the resources of interest. These path expressions represent the precise meaning of its navigation steps through the class (or property) subsumption and/or associations. Furthermore, at each navigation step, the user can extend the generated queries with filtering conditions on the attributes of the currently visited classes and at the same time can easily specify which class of resources should be finally included in the query result. In short, it is an application-independent graphical interface able to generate a unique RQL query which captures the cumulative effect of an entire user navigation session. As an example of a user session with GRQL, consider Figure 2.17 depicting a tree-shaped graphical representation of the class (or property) subsumption relationships defined in an RDFS schema. The user begins his navigation session by choosing one of the available classes (or properties). The selected class (or property) is then displayed in order to allow the user to browse further the directed acyclic graph of class associations. This browsing action is translated into an appropriate RQL query which consists of the specification of the dynamic view over the underlying resource descriptions according to the performed schema navigation. Figure 2.18 shows that class Artist was chosen as entry point along with the generated RQL query. The selection of class Artist returns all the resources classified under Artist and recursively its subclasses (i.e. resources r5 and r6). By providing the definition of the class Artist, the user can now access the resources of other associated classes, as for instance class Artifact (through the created relationship).

2.2.4 OZONE

OZONE [21] stands for a Zoomable Ontology Navigator and it is a tool for searching and browsing ontological information that is defined in DAML (DARPA Agent Markup Language). It reads ontology information and rearranges it visually with context information so that ontology information can be queried and browsed easily and effectively. A visual model is defined for representing the classes, properties and relationships in DAML. The nodes and links as represented in Figure 2.19 are what visually depict the query conditions. Queries can be formulated interactively and incrementally by manipulating objects on the screen. During query formulation, a user can check the intermediate results, which are displayed at the bottom of the

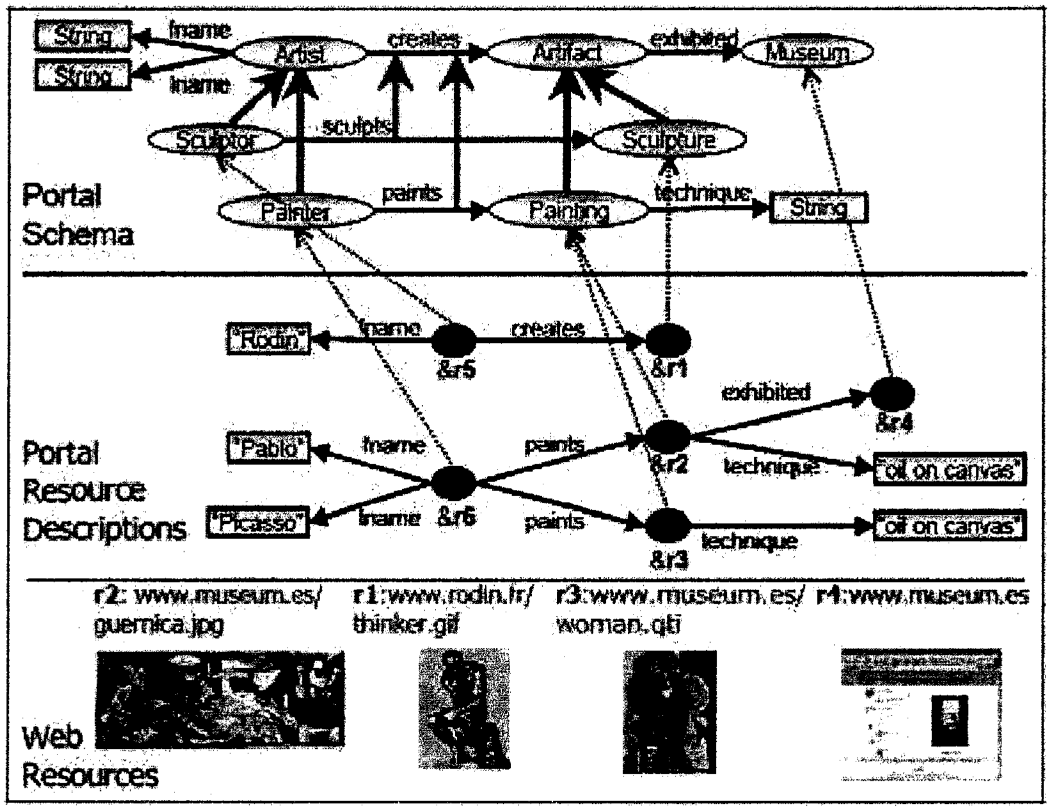
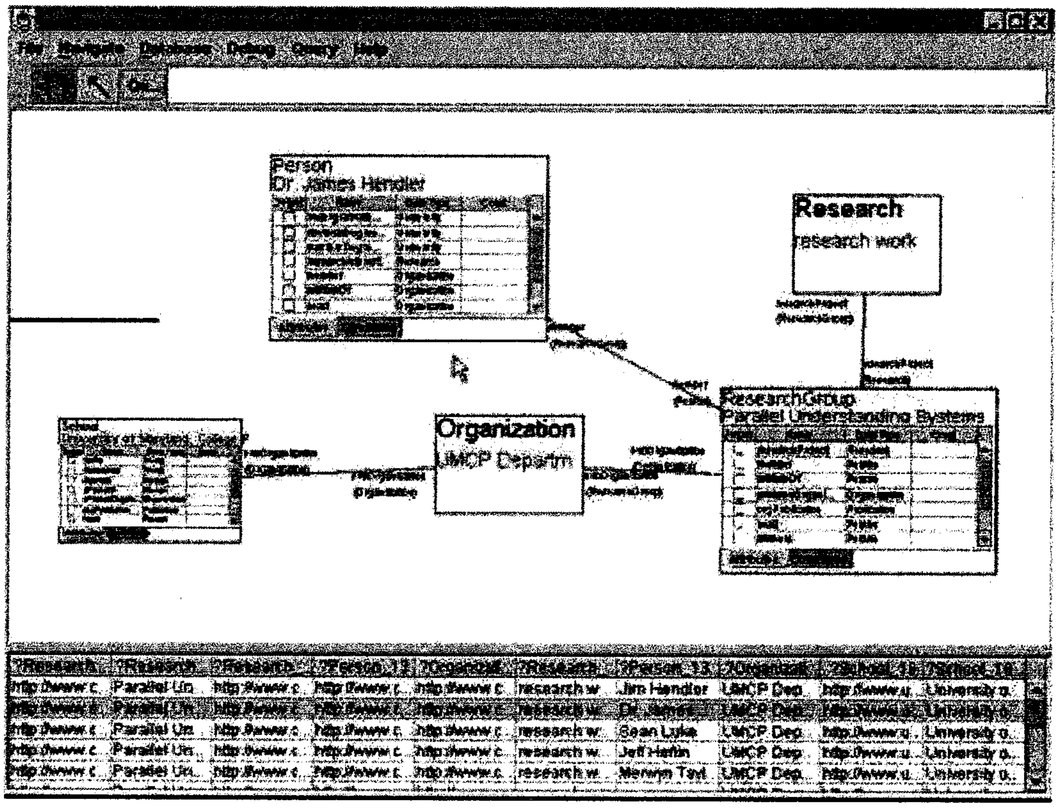


Figure 2.17: RDFS Descriptions of Web Resources in a Cultural Portal. (Figure taken from [2])

Navigation	RQL query	Path
Artist ❶	`select X1 from Artist{X1}`	`Artist`

Figure 2.18: Choice of class Artist as entry point to the schema navigation (Figure taken from [2])



?Research...	?Research...	?Research...	?Person_13	?Organizat...	?Research...	?Person_13	?Organizat...	?School_16	?School_16
http://www.c...	Parallel Un...	http://www.c...	http://www.c...	http://www.c...	research w...	Jim Hendler	UMCP Dep...	http://www.u...	University o...
http://www.c...	Parallel Un...	http://www.c...	http://www.c...	http://www.c...	research w...	Dr. James...	UMCP Dep...	http://www.u...	University o...
http://www.c...	Parallel Un...	http://www.c...	http://www.c...	http://www.c...	research w...	Sean Luke	UMCP Dep...	http://www.u...	University o...
http://www.c...	Parallel Un...	http://www.c...	http://www.c...	http://www.c...	research w...	Jeff Heflin	UMCP Dep...	http://www.u...	University o...
http://www.c...	Parallel Un...	http://www.c...	http://www.c...	http://www.c...	research w...	Merwyn Tayl...	UMCP Dep...	http://www.u...	University o...

Figure 2.19: OZONE overview (Figure taken from [21])

screen. When a result row is selected, each entry in the table is remapped into a corresponding visual node and shown under its title as a blue label [21].

2.3 Summmary

Most of the visual query systems described above visualize a query based on how the data to be queried, whether in a database or an ontology, is visually depicted. As for the database domain, the general trend was to formulate a query on top of the visualization of the database schema. As for the ontology domain, except for GrOWL that permits to query a loaded OWL ontology, most of the designed systems allow to query an ontology that offers an integrated view of multiple database schemas. A description of how the overview of these systems, whether in the database domain or the ontology domain affected and contributed to the design of OntoVQL is presented in Chapter 3, Section 3.2.

Chapter 3

Graphical query language for OWL ontologies

3.1 Background

3.1.1 Brief overview of Description Logics

Description Logics (DLs) is a family of knowledge representation (KR) formalism that describes the knowledge of an application domain (the world) by first defining the concepts (its terminology) of the domain and then using these concepts to specify properties of objects and individuals occurring in the domain (the world description) [3]. The most important features of Description Logic based languages are that they are based on formal, logic-based semantics which can usually be translated into first-order predicate logic and that they offer reasoning services which refer to the knowledge that can be implicitly inferred from the knowledge that is explicitly contained in the knowledge base (KB).

Description Logic supports inferences that are based on the classification of concepts and individuals. The classification of concepts of a given terminology is done by linking these concepts with the subconcept/superconcept relationships (subsumption relationships in DL). This type of relationship allows to structure the terminology in the form of a subsumption hierarchy and thus allows to obtain useful information about the connection of these different concepts. The classification of individuals is done through determining whether a given individual is an instance of a certain

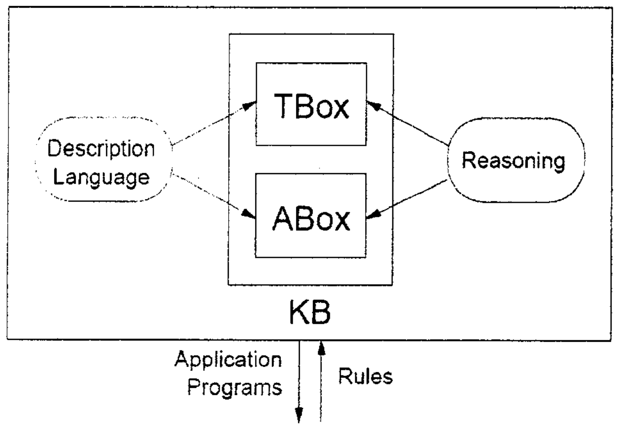


Figure 3.1: Architecture of a knowledge representation system based on Description Logics [3].

concept, that is whether this instance relationship is implied by the description of the individual and the definition of the concept.

The semantics of description logics is defined by interpreting concepts as sets of individuals and roles as sets of pairs of individuals. The semantics of non-atomic concepts and roles is then defined in terms of atomic concepts and roles.

TBox/ABox description

A KR system based on Description Logics provides facilities to set up knowledge bases, to reason about their content, and to manipulate them. Figure 3.1 shows the architecture of such a system.

A knowledge base is composed of two main parts, the TBox and the ABox. The TBox introduces the terminology of the knowledge domain while the ABox contains assertions about named individuals in terms of the terminology described in the TBox.

The terminology in the TBox consists of a set of unary predicate symbols that are used to denote concept names and a set of binary relations that are used to denote role names. Concept names are regarded as atomic concepts, and role names are regarded as atomic roles. In general, a concept denotes the set of individuals that belongs to it, and a role denotes a relationship between instances of concepts. For more complex descriptions, a concept term can be defined recursively from concept names and role names using constructors. Some common constructors include logical constructors in first-order logic such as intersection or conjunction of concepts, union or disjunction of concepts.

A Description Logic system not only stores terminologies and assertions, but also offers services that reason about them. Typical reasoning tasks for a terminology are to determine whether a description is satisfiable (i.e., non-contradictory), or whether one description is more general than another one, that is, whether the first subsumes the second. Important problems for an ABox are to find out whether its set of assertions is consistent, that is, whether it has a model, and whether the assertions in the ABox entail that a particular individual is an instance of a given concept description. Satisfiability checks of descriptions and consistency checks of sets of assertions are useful to determine whether a knowledge base is meaningful at all. With subsumption tests, one can organize the concepts of a terminology into a hierarchy according to their generality. A concept description can also be conceived as a query, describing a set of objects one is interested in. Thus, with instance tests, one can retrieve the individuals that satisfy the query. [4]

3.1.2 Brief overview of OWL ontologies

In both computer science and information science, an ontology is a formal representation of a set of concepts within a domain and the relationships between instances of those concepts. It is used to reason about the properties of that domain, and may be used to define the domain.

Ontologies are used in artificial intelligence, the semantic web, software engineering, biomedical informatics, library science, and information architecture as a form of knowledge representation about the world or some part of it. Common components of ontologies include:

- Individuals: instances or objects (the basic or "ground level" objects).
- Classes: sets, collections, concepts, types of objects or kinds of things.
- Attributes: aspects, properties, features, characteristics, or parameters that are objects (and classes).
- Relations: ways that classes and individuals can be related to one another.

OWL is a Web Ontology language. It is designed for use by applications that need to process the content of information instead of just presenting information to humans [32]. The Web Ontology Language (OWL) is intended to provide a language that can be used to describe classes and relations between individuals that are inherent in Web documents and applications.

An OWL ontology may include descriptions of classes, properties and their instances. In fact, the OWL language is used to formalize a domain by defining classes and properties of those classes, to define individuals and assert properties about them, and to reason about these classes and individuals to the degree permitted by the formal semantics of the OWL language [32].

3.1.3 Brief overview of nRQL

nRQL is an acronym for the new Racer Query Language and has been implemented by an optimized OWL-DL query processor known to be highly effective and efficient [38]. A nRQL query is composed of a query head and of a query body. The query body consists of the query expression. The query head corresponds to the projected variables, that is, variables that are mentioned in the body and will be bound to the Abox individuals that satisfy the query expression. There are many features of nRQL, however only those features that are relevant in the context of this thesis will be covered. An important feature of nRQL is that complex queries are composed from query atoms: concept query atoms, role query atoms, and SAME-AS query atoms.

Query:	(retrieve (?x) (?x |*cat*|))
Result:	(((?x |*Tibbs*|)) ((?x |*Tom*|)))

Table 3.1: Example of a concept query.

These basic expressions that can simply be called atoms are then combined with query constructors, from which only the AND and OR constructors will be covered, to form complex queries. Note that not all nRQL components and constructs are covered. For instance, the SAME-AS query atoms and the negation construct will not be explained since they are not required in the OntoVQL translation to nRQL. Query atoms are either unary or binary in the sense that a unary atom references one object whereas a binary atom references two objects; an object being either a variable or an individual. Also, it is essential to mention that nRQL make use of active domain semantics. This means that variables can only be bound to named individuals occuring in the current ABox.

Concept query

A concept query atom is a unary atom and is used to retrieve the instances that belong to a concept or an OWL class. The query in Table 3.1 is asking to retrieve the individuals that are member of the cat concept in the people-pets.owl (see appendix) ontology. The answer to the query is a list of variable-value pairs where the variable ?x is bound to Tibbs and Tom. Thus, the set-up of the result is specified by the head of the query which is (?x). The query body or query expression is enclosed by the second pair of brackets and consists of a single query atom.

Role query

A role query atom is a binary atom and is "used to retrieve pairs of role fillers from an Abox, or pairs of OWL (RDF) individuals" [38]. The body of the first query in Table 3.2 is a role query atom whose meaning is to find pet owners and their pets. Again, the result is a list of instances bound to the variables determined by the query

Query 1:	(retrieve (?x1 ?x2) (?x1 ?x2 |has-pet|))
Result 1:	(((?x1 |*Minnie*|) (?x2 |*Tom*|)) ((?x1 |*Walt*|) (?x2 |*Louie*|)) ((?x1 |*Walt*|) (?x2 |*Dewey*|)) ((?x1 |*Walt*|) (?x2 |*Huey*|)) ((?x1 |*Mick*|) (?x2 |*Rex*|)) ((?x1 |*Fred*|) (?x2 |*Tibbs*|)) ((?x1 |*Joe*|) (?x2 |*Fido*|)))
Query 2:	(retrieve (?pets) (|*Minnie*| ?pets |has-pet|)))
Result 2:	(((?PETS |*Tom*|)))

Table 3.2: Example of role queries.

head. A more specific query can be formulated by setting the domain or range with an instance. Suppose the objective was to find pets of Minnie. Then, the domain variable ?x1 is replaced with the individual Minnie as shown in the second query of Table 3.2, and thc result is therefore reduced to only one tuple. Note that since nRQL employs active domain semantics, then only those role fillers that are named individuals in the ABox are retrieved. For example, if a pet owner is defined in the TBox as a person who has a pet and the ABox contains an individual Peter defined as a pet owner but not modeled as a role filler for the has-pet role, then Peter will not be returned in the result for the role query atom (?x ?y has-pet).

The AND constructor

The AND constructor is used in order to express a conjunction. A classic example would be to conjunct concepts query atoms with role query atom as shown in the first query of Table 3.3. In this case, the role fillers are constrained to belong to the female class for the domain and to the cat class for the range of the role 'has-pet'. A variable is used in more than one atom for the purpose of identifying the set of individuals that satisfy the same constraints. Thus, an instance bound to ?x1 must fulfill the conjunction of the constraints of being a female and petting a cat which is found to be Minnie. Another example that demonstrates the importance of the variable assignment while using the AND constructor is shown by comparing the second and third query. Both queries are identical except that in the second query the same variable is assigned to all concepts whereas in the third query, distinct variables are assigned. Because of the difference in the variable assignment, the query semantics are different as well. The meaning of the second query is to find individuals that satisfy all the constraints in the query body, that is "adults who like dogs and own a dog". On the other hand, the meaning of the third query is to identify the instances that are modeled as adults, those that like dogs and those that own a dog. As a consequence, the dissimilarity in the semantics yields different results. Therefore, in order to achieve the true semantics of query conjunction, that is set intersection semantics, it is required to assign the same variable to constraints.

The OR constructor

The OR constructor allows to have disjunction. Its semantics consist of unifying the answer set of the OR constructor arguments. The issue of variable assignment mentioned earlier for the AND constructor needs to be examined for the OR constructor as well because variable naming directly impacts the query results in this case. In fact, when different variables are used in the arguments of a disjunction, nRQL ensures that each of these arguments references the same variables. To illustrate this by an example, consider the first query in Table 3.4. This query is actually rewritten by nRQL into

```
(retrieve (?x ?y) (or (and (?x cat) (?y top)) (and (?x top) (?y dog))))
```

In the first argument, instances that belong to the cat concept. Tibbs and Tom,

Query 1:	(retrieve (?x1 ?x2) (and (?x1 |*female*|) (?x2 |*cat*|) (?x1 ?x2 |has-pet|)))
Result 1:	(((?X1 |*Minnie*|) (?X2 |*Tom*|)))
Query 2:	(retrieve (?x) (and (?x |*adult*|) (?x |dog-liker|) (?x |dog-owner|)))
Result 2:	(((?X |*Mick*|)))
Query 3:	(retrieve (?x1 ?x2 ?x3) (and (?x1 |*adult*|) (?x2 |dog-liker|) (?x3 |dog-owner|)))
Result 3:	(((?X1 |*Minnie*|) (?X2 |*Joe*|) (?X3 |*Mick*|)) ((?X1 |*Minnie*|) (?X2 |*Mick*|) (?X3 |*Joe*|)))

Table 3.3: Example of queries with the AND constructor.

are bound to ?x while any other instance is bound to ?y. This is because 'top' is the super concept for all concepts and thus all instances belong to the 'top' concept. On the other hand, in the second argument, instances that are members of the dog concept, Rex and Fido, are bound to ?y while any other instance is bound to ?x. Therefore, the result of the first query is the union of the answer set of (retrieve (?x ?y) (and (?x cat) (?y top))) and of (retrieve (?x ?y) (and (?x top) (?y dog))). However, when the same variable is assigned to both arguments in the disjunction, the result reflects the semantics of set union. Hence again, to obtain a meaningful result with the OR constructor, arguments must reference the same variable. Note that in order to respect this condition; these arguments must also be of the same arity. For instance, if a role atom referring to ?x and ?y and a concept atom referring to ?x are unified, then the query will be rewritten so that each argument refers to both ?x and ?y.

Variable types

There exist two types of variables in nRQL: injective and ordinary (non-injective). An injective variable is prefixed by "?", as it is the case in all the above queries, and is an indication for injective mapping from variables to ABox individuals. In an injective mapping, an injective variable can only be bound to an ABox individual if no other injective variable is bound to it. For example, if ?x is bound to Tom, then ?y cannot be bound to Tom as well. On the other hand, an ordinary variable is prefixed by "\$?" and is an indication that the mapping is not necessarily injective, i.e. it can be an arbitrary mapping where \$?x and \$?y are both bound to Tom. The choice of the variable type in a query may have an impact on the result. The example in Table 3.5 is an illustration where this in effect takes place. Both queries are identical except for their variable types, the first having injective variables and the second ordinary ones. By analyzing the outcomes, the first thing to notice is that the cardinality of the result for the second query surpasses that of the first. Furthermore, the additional elements

Query 1 and Query 2 are rewritten as	(retrieve (?x ?y) (or (and (?x cat) (?y top)) (and (?x top) (?y dog))))
Query 1:	(retrieve (?X ?Y) (or (?X |*cat*|) (?Y |*dog*|)))
Result 1:	(((?X |*Tibbs*|) (?Y |*Minnie*|)) ((?X |*Tibbs*|) (?Y |*Tom*|)) ... ((?X |*Tom*|) (?Y |*Minnie*|)) ((?X |*Tom*|) (?Y |*Walt*|)) ... ((?X |*Minnie*|) (?Y |*Rex*|)) ((?X |*Walt*|) (?Y |*Rex*|)) ... ((?X |*Minnie*|) (?Y |*Fido*|)) ((?X |*Walt*|) (?Y |*Fido*|))
Query 2:	(retrieve (?X) (or (?X |*cat*|) (?X |*dog*|)))
Result 2:	(((?X |*Tibbs*|)) ((?X |*Tom*|)) ((?X |*Rex*|)) ((?X |*Fido*|)))

Table 3.4: Example of queries with the OR constructor.

Query:	(retrieve (?x ?y ?z) (and (?x ?y |has-pet|) (?x ?z |likes|)))
Result:	(((?X Walt)(?Y Louie)(?Z Dewey)) ((?X Walt)(?Y Louie)(?Z Huey)) ((?X Walt)(?Y Dewey)(?Z Huey)) ((?X Walt)(?Y Dewey)(?Z Louie)) ((?X Walt)(?Y Huey)(?Z Dewey)) ((?X Walt)(?Y Huey)(?Z Louie))) 6 elements
Query:	(retrieve ($?x $?y $?z) (and ($?x $?y \|has-pet\|) ($?x $?z |likes|)))
Result:	((($?X Walt) ($?Y Huey) ($?Z Dewey)) (($?X Walt) ($?Y Huey) ($?Z Louie)) (($?X Walt) ($?Y Dewey) ($?Z Huey)) (($?X Walt) ($?Y Dewey) ($?Z Louie)) (($?X Walt) ($?Y Louie) ($?Z Huey)) (($?X Walt) ($?Y Louie) ($?Z Dewey)) (($?X Walt) ($?Y Louie) ($?Z Louie)) (($?X Walt) ($?Y Dewey) ($?Z Dewey)) (($?X Walt) ($?Y Huey) ($?Z Huey)) (($?X Minnie)($?Y Tom) ($?Z Tom)) (($?X Mick) ($?Y Rex) ($?Z Rex)) (($?X Fred) ($?Y Tibbs) ($?Z Tibbs)) (($?X Joe) ($?Y Fido) ($?Z Fido))) 13 elements

Table 3.5: Query examples with different variable types.

in the second outcome are partly due to the different combinations where the second and third variables of the result elements are bound to the same individual, and due to new result elements that were not part of the first query's answer. There were four instances (Minnie, Mick, Fred and Joe) that liked and pet a single ABox individual and this caused these instances to be omitted from the first result because ?y and ?z cannot be bound to the same individual. Therefore, although that the answer to the first query is semantically complete, from the users point of view it would seem 'incomplete' because they would have expected to find 'Minnie, Mick, Fred and Joe' in the result as they are present in the answer to the second query.

3.2 Contribution of Literature review to OntoVQL

Performing an overview of what has been achieved on query visualization and reviewing the different approaches that were adopted for query representation played a major role in tracking down and coming up with the essential characteristics that should be part of OntoVQL. In summary, the guidelines behind OntoVQL are the following:

- should provide an alternative to textual languages
- should be simple to explain to users, the query should be expressed in a single diagram that can be broken into parts of distinct queries and recombined later
- should be formal
- should express the basic querying operators of conjunction and union.

It is important to highlight the reasons behind these guidelines. First, let us examine why a VQL should provide an alternative rather than replace a textual language. From all the visual query languages that were reviewed, none of them had the full expressive power of a textual query language. It is not possible to visually depict the totality of language constructors in a simple straightforward way. In fact, expressiveness is inversely proportional to clearness. Therefore, a VQL can obviously not replace a textual query language. On the other hand, it can become a valuable alternative providing that it is easy to grasp. More specifically, for a subset of the textual query language, the VQL must provide visualizations that mask the complexity of the syntax and thus for the same degree of expressivity offer a simpler way of querying. It goes without saying that a VQL that effectively hides the complexity of a subset of a textual query language but at the same time is not easily comprehended by a user is not desirable since this would contradict the very reason of why we are introducing this new VQL. Therefore, it is important that the

VQL employs common symbols or notations that are widely used and thus easily understood by most people.

In that sense, one would think that the use of icons would be a great way of incorporating this idea in the VQL as it was the case for Klaeidoquery reviewed in Section 2.1.3. Nonetheless, the use of icons was omitted in designing OntoVQL because although icons have their set of advantages, they do not come without some drawbacks. In fact, icons cannot be generic and have to be domain specific. Actually, for icons to be meaningful, it is ideal that domain specific experts design them since this would help to recreate the environment that they are familiar with [25]. Therefore, if we were to adopt icons in OntoVQL, we will be facing the problem of getting restricted to choose one specific domain and the user will loose the liberty of loading any OWL ontology into the system for querying it.

Since VQLs are mainly based on the idea of directly manipulating a database visualization as a graph by selecting the parts that are to be included in the query, then by analogy, given that the TBox of an ontology can also be visually represented by a graph, it is logical to consider that in order to formulate a query for ABox retrieval, one has to select "parts" of the TBox, i.e. concepts and roles, as components of the query. The query is thus composed of graph components and therefore can be formulated as a graph on its own. This would be one way of understanding the logistics behind viewing an ABox query as a graph. The other way is based on the nature of ABox querying itself. For instance, the starting point for ABox retrieval is the extraction of all instances belonging to a concept. Then, one step further from getting a set of individuals based on a concept name is to learn how these individuals are related with other individuals by the means of a role represented as an edge. In that sense, it is natural to construct query constraints in the form of a graph.

Another major argument to adopt diagrammatic and graph-based languages is that they are easier to formalize than other approaches such as the form-based or

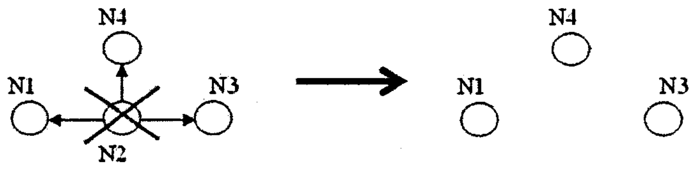


Figure 3.2: Breaking down a query by deleting a node.

iconic-based ones [1]. In fact, graphical languages are more suited to be formalized, given the precise mathematical structure (i.e. graph) the diagrams are based on [10]. Configuring a query as a graph is not only convenient for the sake of its representation but also for its manipulation. Actually, since each graph stands for a single query, combining two graphs with an edge would stand for combining two queries and thus creating a new one. Also, deleting a component from the graph such as a node would cause the graph to break down into several different queries. In Figure 3.2, deleting node N2 had as a repercussion the deletion of all its edges resulting in the creation of three new queries N1, N3 and N4. The ability of combining queries to form a new complex one and of breaking down a complex query into several simpler ones was thought to be a powerful way of allowing the user to fully appreciate the advantage of using a VQL instead of a textual query language. In fact, when a querying session permits the user to view multiple queries simultaneously and manipulate them to form a new query, this clearly gives much more flexibility in the query formulation process in comparison to a typical textual query language for which each submitted query cannot be related to the other submitted ones.

Finally, the last guideline point stated above mentions that OntoVQL should include the querying operators of conjunction and union. The ideal situation would have been to incorporate all or most of nRQL (the New RacerPro Query Language) [18] operators within OntoVQL. However, because of time constraints and difficulty encountered in visualizing them, the decision was made to only design conjunction and union because these two operators can be considered to be the basic ones in nRQL that allow expressing large complex queries.

3.3 Visual notations

The ontology Abox primarily contains concept and role assertions. In order to retrieve instances that have been asserted to a concept or a role, a query would necessitate either one or both of the two types of constraints. The first constraint is to specify a concept. A Concept specification restricts the result to the instances that are bound to the specified concept. The second constraint is to specify a role for which instances are fillers. In that case, the result consists of pairs of individuals that are related by the specified role. In a query visualized by a graph, that we will call a "graph query", a vertex would stand for a concept specification and a directed edge for a role linking two instances. In order to remain visually consistent with respect to previous visualizations, concept and role constraints are represented following the conventions described in [9]. Thus, a concept name inside a filled oval (see Figure 3.3(a)) semantically maps to a concept constraint whereas a role name and an arrow pointing from one "entity" to another one correspond to a role constraint (see Figure 3.3(b)). A role is meant to connect different types of nodes in the 'graph query' other than the concept constraint type. Actually, one of the cases requiring a different type of vertex would be in the event we are looking for pairs of instances related by a certain role without specifying any constraints for the domain and/or range nodes. These vertices would then be considered as "unknown" indicating that the instance 'identity', that is the concept(s) to which the instance belongs, is unknown. This type of vertex is most conveniently represented by a '?' in a filled oval (see Figure 3.3(c)). Note that what is at stake in this representation is only the shape whereas the color is of no importance. Another type of vertex would be the "instance node". In that case, either the domain or the range is restricted to be bound to a specific instance. In other words, the query is about finding all those instances related to instance 'A' by role

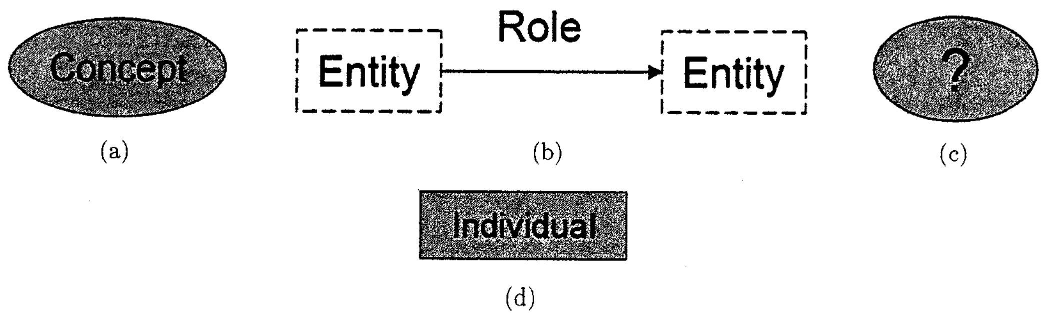


Figure 3.3: Visualization of the basic elements of OntoVQL.

'B'. An instance vertex is represented by the instance name in a filled rectangle (see Figure 3.3(d)). Note that an instance node cannot exist in a query by itself and must be part of a role constraint. The same can be said for the unknown node. However, this is not the case for the concept node since it represents a constraint on its own.

The operators used for assembling building blocks or connecting query constraints into a query are intersection and union. At the most basic level, these operators are applied to concepts. The intersection of concepts C1, C2, . . . Cn in a query means that an instance must belong to all these concepts while their union means that an instance must belong to at least one of them. Even though the semantics is different, their representation is alike except for the identifying keyword 'AND' for intersection and 'OR' for union. A node with the AND/OR keyword connected by edges to the concepts that are to be intersected or unified represents concept intersection/union (see Figure 3.4(a)). However, this representation can be simplified. In effect, it is more suitable to represent these operations by a circle labeled with the AND/OR keyword that encloses the intersected or unified concepts (see Figure 3.4(c)). As an example of how this simplification takes effect, compare the two queries in Figure 3.4(b, d). Note that although both graph queries have the same meaning, the second one is much simpler in view of the fact that it got rid of four vertices and two edges with respect to the first one. Consequently, the second representation was adopted for viewing the intersection/union of concepts. This OntoVQL constructor is called the AND/OR

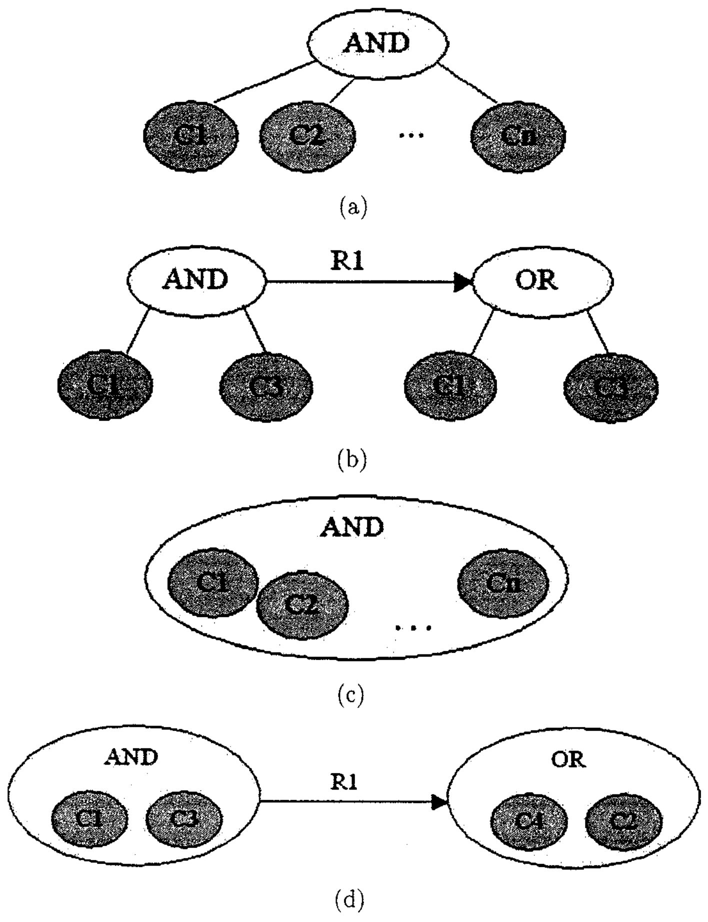


Figure 3.4: Representation of concept intersection.

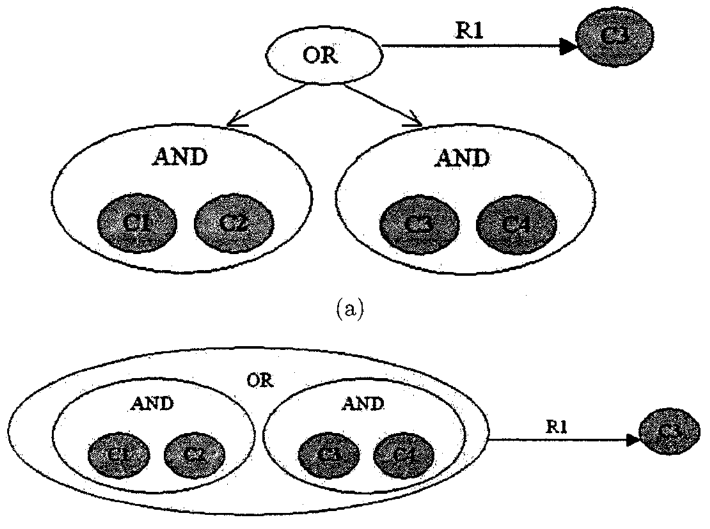
group. An AND/OR group is considered to be another vertex type and thus when a role constraint is added to the group, it is added to the group as a whole and not specifically to one of the concepts inside the group.

Now that the representation of intersection/union of a group of concepts is settled, we are left with the issue of how to represent the intersection/union of these groups. The first idea would be to follow the same approach as before, that is enclosing AND/OR groups inside a circle with the AND/OR keyword indicating whether it is an intersection or a union. However, even though this seems logical, there exists a major drawback to it. Actually, this visual simplification restrains the expressive power of the VQL. This is best explained by taking a look at the following example.

In Figure 3.5, the first graph query is depicted according to the first approach where the OR node is linked to the AND groups with directed edges, whereas the second graph query is the "simplified" version of the first since the AND groups and OR node were assembled into one entity. This simplification is traded with the impossibility of linking the AND groups to other nodes since they are no longer a node by themselves and were engulfed within another entity. Therefore, intersection and union of AND/OR groups is best represented by the first approach and thus a new type of node is added to OntoVQL: the AND/OR node. Notice that in Figure 3.5(a), the edges that link the OR node to the AND groups are directed edges. The reason for that is to avoid semantic confusion. In fact, when a query has several levels of alternating union and intersection as shown in Figure 3.6, it may become confusing as how to interpret it. At first view, both queries in Figure 3.6 seem to be dissimilar but as a matter of fact, they are identical except that they are depicted in two distinct "layouts". The difference in layout resides in having the first one with root node number 1 whereas the second one has root node number 2. These two layouts semantically translate into two different logical sentences. As a result, the same graph query would have more than one meaning. This ambiguity is removed by introducing the directed edge as a component that connects the AND/OR node to the node taking part in the intersection/union. In this fashion, no matter how the OntoVQL components are placed, there can be only one way to translate the graph query as it is shown in Figure 3.6 where despite the two different layouts, both graph queries are identical in terms of having the exact same visual constructs and semantics.

3.4 Visual Syntax

In the previous section, all the components of OntoVQL were introduced. In total, they were eight from which concept, unknown concept, individual, AND/OR group, AND/OR node were vertices and two of them, role and connection edge, were edges



(b)

Figure 3.5: The same query represented in two different graphs.

linking these vertices. How these components should be connected together constitute the "connectivity syntax" of OntoVQL. The syntax was established based on the need of producing a graph query that is unambiguous, in the sense that it can be interpreted in only one way, and is easily comprehensible. In order to achieve that goal, there are few constraints that were introduced and which we will informally cover at first. The simplest way of constructing a visual query is to either have a single vertex on its own as a concept query or to have a couple of vertices connected by roles which is a role query. The shape of a role query produces a graph that is either cyclic or acyclic as shown in the graph query examples of Figure 3.7(a,b). An imposed restriction on the role query is that there can only be one role between any given pair of vertices for the simple reason of prioritizing visual clarity. Furthermore, more complicated queries would involve the alternation of union and conjunction operators. Such queries result in a graph that has the form of a tree, i.e. a node with an arbitrary number of branches connecting to other nodes, where the root is an AND/OR node and the leaves are AND/OR groups (Figure 3.7(c)). Both of these shapes (acyclic/cyclic graph and tree) can exist together in a graph query as it is shown in Figure 3.7(d). Note that query

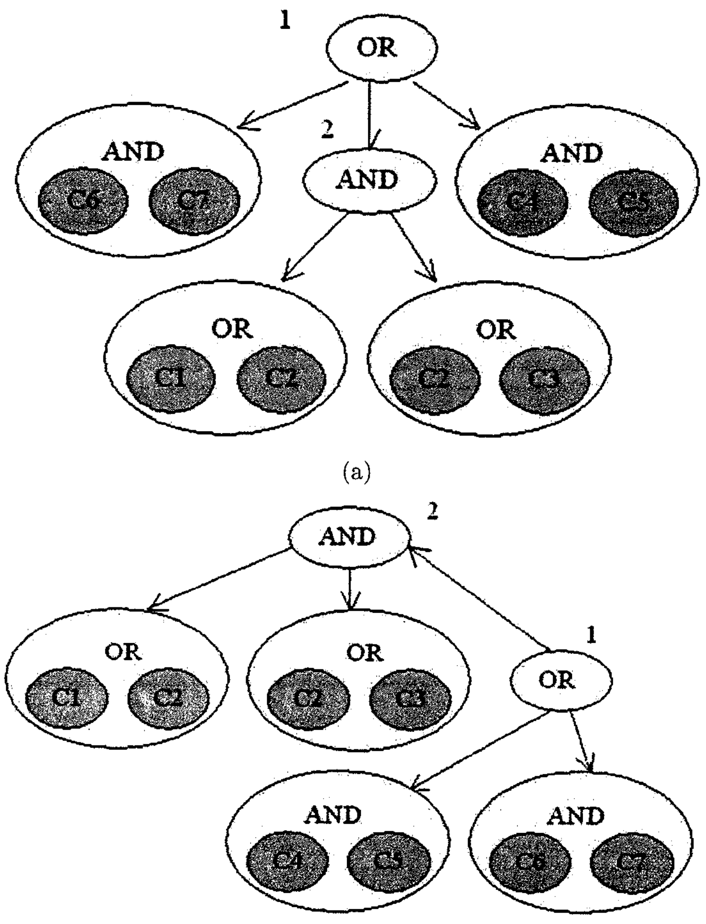


Figure 3.6: The same query in two different layouts: (a) (OR (AND C6 C7) (AND C4 C5)(AND (OR C1 C2) (OR C2 C3))) (b) (AND (OR C6 C7) (OR C4 C5)(OR (AND C1 C2) (AND C2 C3)))

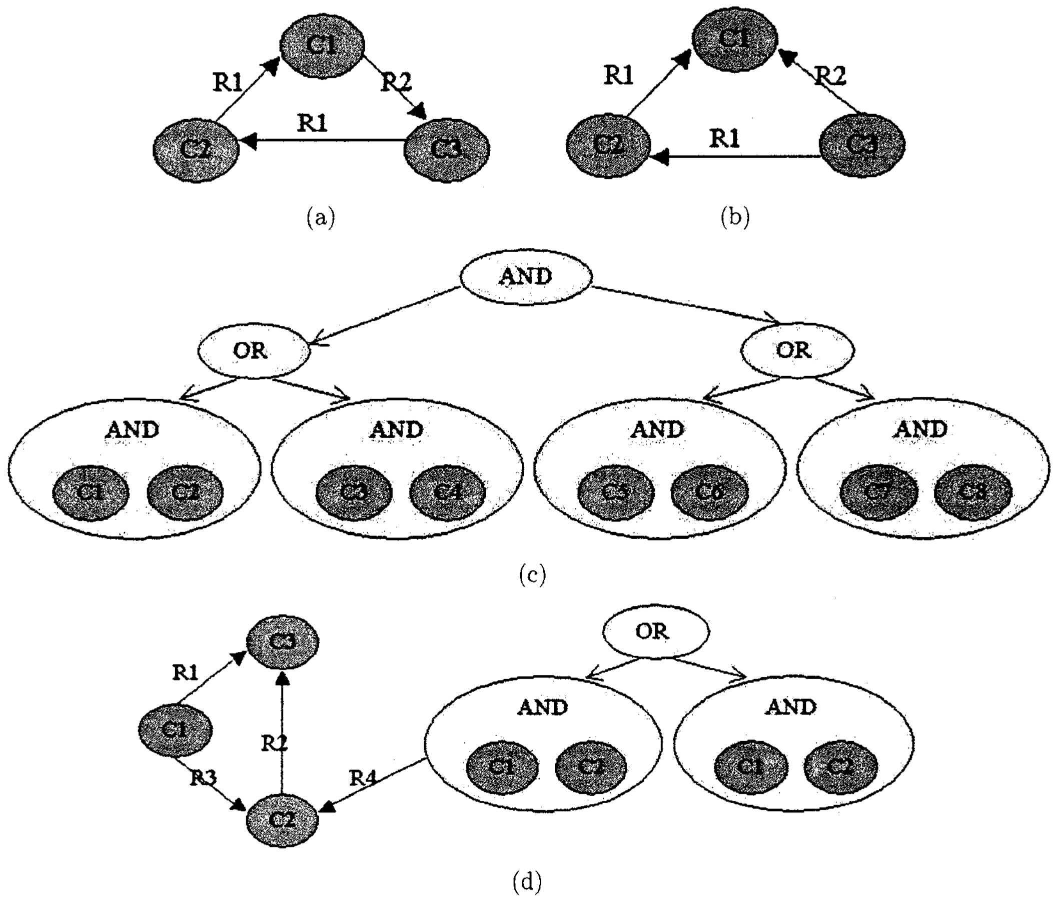


Figure 3.7: Shapes in a graph query: (a) Example of a cyclic graph query. (b) Example of an acyclic graph query. (c) Example of a tree graph query. (d) Example of a graph Query with tree and cyclic graph shapes.

expressions are compositional and their logical structure is not flat but tree shaped [11] and thus it was natural to adopt this structure in the graph representation of the query.

There are two rules concerning the visual connectivity of OntoVQL that are responsible for enforcing the tree shape in a graph query. The first rule consists of not allowing an AND/OR node or group to have more than one incoming connection edge. In Figure 3.8, the graph query on the top has an AND group with two incoming connection edges. The meaning can be understood as the union of the three AND groups. Therefore, the second OR node is clearly superfluous. The graph query on

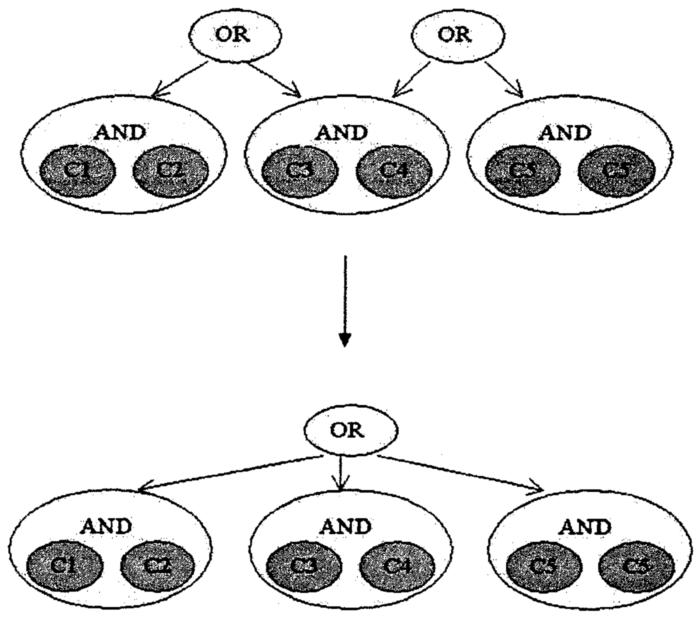
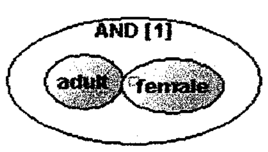


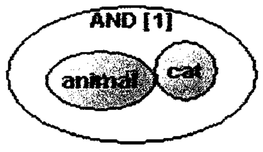
Figure 3.8: Simplification of a graph query by applying the first visual constraint.

the bottom is the result of applying the first rule that got rid of the extra OR node and by the same occasion gave the graph its tree form. Thus, this rule was adopted in order to avoid redundancy and to simplify the graph for more clearness of the query semantics (Figure 3.8).

The second visual constraint enforcing the "tree shape" on a graph forbids the existence of a role between two entities that are not under the same "branch" of a tree in the graph query. The word "branch" is used to refer to the tree divisions in the graph. For instance, in Figure 3.7(d), concepts C1, C2 and C3 are under the first "branch". In order to give a thorough explanation of what the second constraint means and why it was established, we will go through a series of examples. Let us perform some queries on the people-pets ontology for that purpose. First, we start with two simple queries: two AND groups (Figure 3.9). We learn from the result that only Minnie is a female adult and that there are only two cats: Tibbs and Tom. Then, connecting the AND groups with the "has-pet" role, we get that Minnie has



(a)



(b)

Figure 3.9: AND group queries (a) The result from Racer for this query is: ((?X1 |*Minnie*|)) (b) The result from Racer for this query is: ((?X1 |*Tibbs*|) (?X1 |*Tom*|))

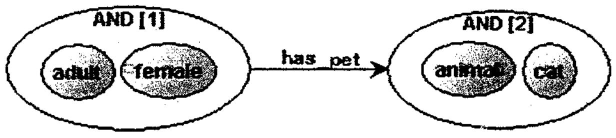


Figure 3.10: A query composed of two AND groups linked with a role.

Tom as a pet (see Figure 3.10, Table 3.6 for translation, Table 3.7 for result).

Another possibility for connecting these two AND groups other than with a role is with connection edges coming from an OR node (Figure 3.11) which means that we are looking for the union of these AND groups. In that case, as expected, the answer is Minnie, Tibbs, and Tom. What happens if we connect these two groups with a role and with an OR node at the same time becomes rather confusing (Figure 3.12). The trouble with this graph query is that an element from one "branch" is connected to an element of another "branch". There are several problems with such a graph query. Let us mention beforehand that the explanation given here may be better understood after reading Chapter 3.4 that covers the mapping of OntoVQL into nRQL. However, what is at stake for the present argument is that each vertex must be bound to a variable when the graph query is translated. As explained later in Chapter 3.4,

```
(retrieve ( ?x2 ?x1 )
          (and (and (?x2 |animal|)
                    (?x2 |cat|))
               (and (?x1 |adult|)
                    (?x1 |female|))
          (?x1 ?x2 |has-pet|)))
```

Table 3.6: Translation of query in Figure 3.10 to nRQL.

((?X1 $|Minne|$) (?X2 $|Tom|$))

Table 3.7: Result from Racer for query in Figure 3.10

an appropriate mapping of the OR node requires that whatever is connected to the node must be bound to the same variable. However, in that case the domain and range of the role would be bound to the same variable as well. This is a problem because the role component implies distinct variables for the domain and range while the OR constructor implies a single variable. Which semantics should be applied? Furthermore, the graph query meaning is ambiguous and leads to confusion. What are we really implying to look for with this query? Are we looking for the union of individuals that are either in the domain ('adult' ⊓ 'female') or the range ('animal' ⊓ 'cat') of the "has-pet" role (let us label this as case 1)? Or, are we looking for individuals that are instances of (('adult' ⊓ 'female') ⊔ ('animal' ⊓ 'cat')) and are fillers for the "has-pet" role (let us label this as case 2)? If the implied meaning was as in case 1, then the nRQL translation in Table 3.10 does not correspond to the implied semantics as it can be concluded from the obtained result "Minnie, Tibbs, Tom". In fact, the correct expected answer should be "Tom, Minnie". However, if roles were not permitted between elements of different branches and we were to represent the semantics of case 1, then the visual query would look like in Figure 3.13. This visualization is not ambiguous since it can have only one possible meaning and its translation to nRQL corresponds to the visually implied semantics. On the other hand, if the implied meaning is as in case 2, then the translation (Table 3.10) of the graph query in Figure 3.12 does not correspond to this semantics either. This is because by binding the role's domain and range to the same variable implies to find individuals related to themselves. In fact, the last line of the nRQL translation of this query in Table 3.10 is looking for pairs of instances such as "instance A-has-pet-instance A". Obviously, the latter implication does not correspond to the graph

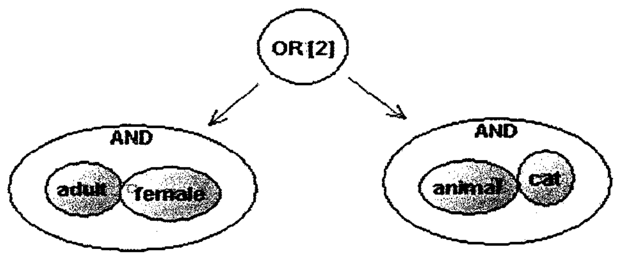


Figure 3.11: Query composed of an OR node linking two and groups.

```
(retrieve ( ?x2 )
          (and (or (and (?x2 |adult|)
                        (?x2 |female|))
                   (and (?x2 |cat|)
                        (?x2 |animal|)))))
```

Table 3.8: Translation of query in Figure 3.11 to nRQL (URLs are omitted for clarity)

query in Figure 3.12. However, it matches the one in Figure 3.14 whose visual meaning conforms to the case 2 semantics. Thus, once again, by not connecting the two entities belonging to different "branches" with the "has-pet" role, the graph query is cleared from its ambiguity since the graph query in Figure 3.14 can have only one possible interpretation.

Another essential rule that needs to be introduced consists of not allowing an entity to belong to more than one "branch" in the tree shape query because that would make the translation of the visually implied semantics impossible. In order to thoroughly understand this rule, consider the examples in Figure 3.16. The graph query contains a concept that is found to belong to two "branches" at the same time. The visual semantics of this query implies that the range of the first and second "is-pet-of" role is the same instance. Consequently, the rightful expectation would be that the tuples (animal → person) and (duck → person) would be merged only when

```
((?X1 |Tom|)
 (?X1 |Minnie|)
 (?X1 |Tibbs|))
```

Table 3.9: Result from Racer for query in Figure 3.11

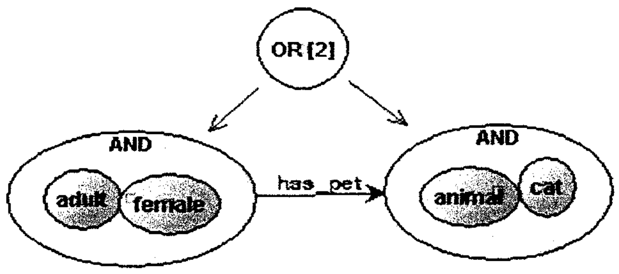


Figure 3.12: Example of an illegal query.

```
(retrieve ( ?x2 )
          (and (or (and (?x2 |adult|)
                         (?x2 |female|))
                    (and (?x2 |cat|)
                         (?x2 |animal|))
               (?x2 ?x2 |has-pet|))))
```

Table 3.10: Translation of query in Figure 3.12 to nRQL

```
(((?X2 |Minnie|))
 ((?X2 |Tibbs|))
 ((?X2 |Tom|)))
```

Table 3.11: Result from Racer for query in Figure 3.12

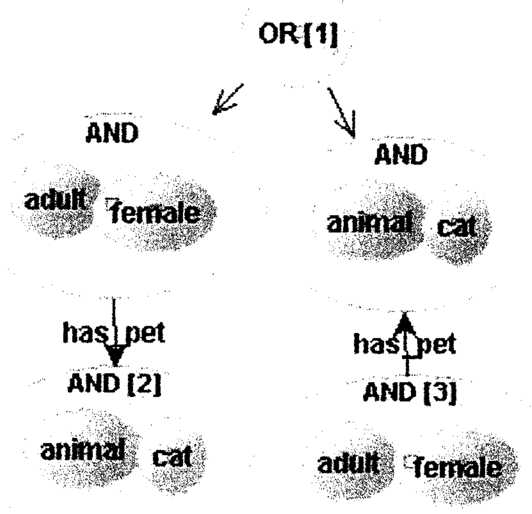


Figure 3.13: Correct representation of the query in Figure 3.12.

```
(retrieve ( ?x5 )
          (and (or (and (?x5 |cat|)
                        (?x5 |animal|)
                        (and (?x6 |adult|)
                             (?x6 |female|))
                        (?x6 ?x5 |has-pet|))
                    (and (?x5 |female|)
                         (?x5 |adult|)
                         (and (?x7 |cat|)
                              (?x7 |animal|))
                         (?x5 ?x7 |has-pet|))))))
```

Table 3.12: Translation of query in Figure 3.13 to nRQL

```
((?X1 |Tom|)
 (?X1 |Minnie|))
```

Table 3.13: Result from Racer for query in Figure 3.13

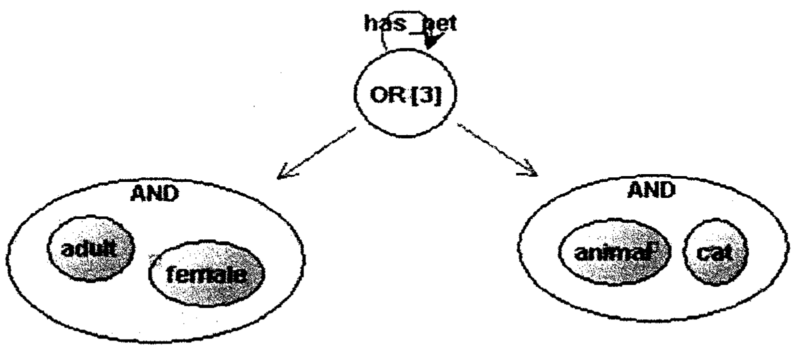


Figure 3.14: Correct representation of the query in Table 3.10.

```
(retrieve ( ?x3 )
          (and (?x3 ?x3 |has-pet|)
               (or (and (?x3 |female|)
                        (?x3 |adult|))
                   (and (?x3 |animal|)
                        (?x3 |cat|)))))
```

Table 3.14: Translation of query in Figure 3.14 to nRQL

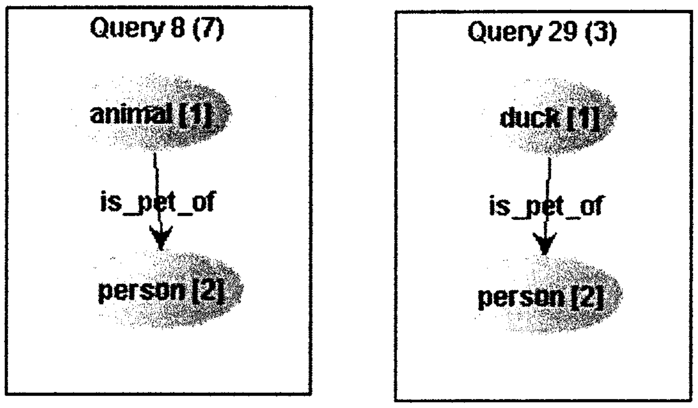
```
(retrieve ($?x2 $?x1)
          (and (or (and ($?x2 |person|)
                        (and ($?x1 |animal|))
                        ($?x1 $?x2 |is-pet-of|))
                   (and ($?x2 |person|)
                        (and ($?x1 |duck|))
                        ($?x1 $?x2 |is-pet-of|)))))
```

Table 3.15: Translation of query in Figure 3.16 to nRQL.

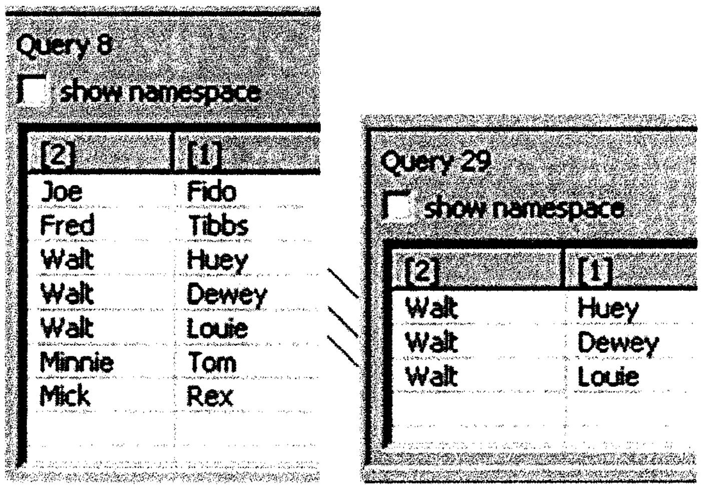
$?x1	$?x2
Huey	Walt
Dewey	Walt
Louie	Walt
Tibbs	Fred
Fido	Joe
Rex	Mick

Table 3.16: Result from Racer for query in Figure 3.16 contains 7 tuples

the person concept is bound to the same individual. Therefore, the expected answer when taking a look at the results in queries 8 and 29 in Figure 3.15 would include only the tuples for query 29. In order to get such a result, one would think that what is needed is to assign the same variable to the person concept for both branches as is shown in the nRQL translation. However, when submitting the query to Racer, the actual result does not match the expected one. In fact, the result consists of the tuples that were obtained for query 8 instead of those for query 29. This shows that assigning the same variable to more than one concept under the AND or OR construct in the nRQL translation does not indicate that in the underlying visual query this concept is shared by more than one "branch". In other words, even though the translation of the query binds the concept person to the same variable, the result does not match the visual query semantics. Then, this example also shows that it is impossible to translate such visual semantics to nRQL. Therefore, a better approach is to represent this query as in Figure 3.17. This query gets rid of the common concept between the two branches. In this way, the visual ambiguity from the graph query in Figure 3.17 is cleared.



(a)



Query 8

show namespace

[2]	[1]
Joe	Fido
Fred	Tibbs
Walt	Huey
Walt	Dewey
Walt	Louie
Minnie	Tom
Mick	Rex

Query 29

show namespace

[2]	[1]
Walt	Huey
Walt	Dewey
Walt	Louie

(b)

Figure 3.15: (a) Two role queries in OntoVQL. (b) Results of the queries.

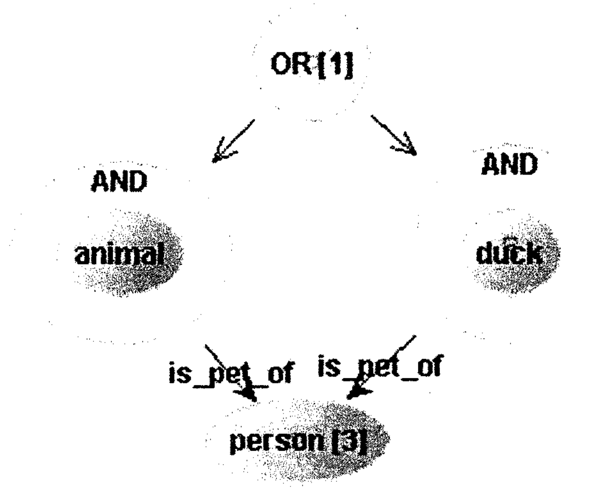


Figure 3.16: Example of an illegal graph query where an entity (person concept) is under more than one "branch".

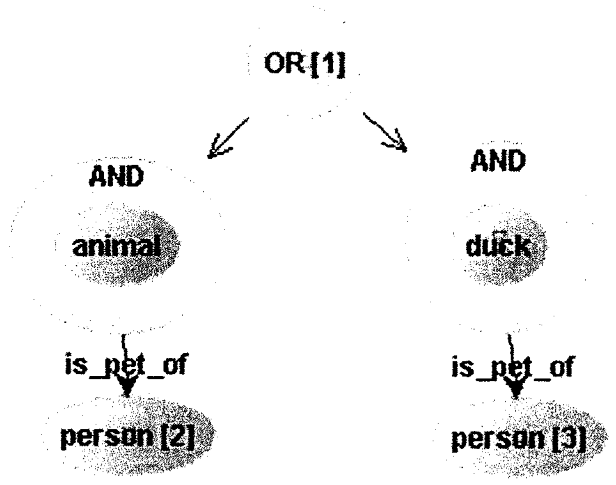


Figure 3.17: Correct representation of the query in Figure 3.16.

Now that we have looked in an informal way how OntoVQL components can be combined to produce meaningful queries, we will encode these assumptions into a formal grammar. In linguistics and computer science, a generative grammar is a formal grammar that provides a precise description of a formal language. The grammar is composed by a set of rules dictating the syntax of the language, i.e., expressing how strings in a language are generated. We have adapted this principle in the context of generating a visual language instead of a textual one in order to generate OntoVQL by a formal grammar or more precisely by an adjusted version of a formal grammar. As a result, since OntoVQL can be generated by a formal grammar and is also semantically and syntactically unambiguous, it can be viewed as a formal visual query language. As previously stated, what is of primary importance that has a direct impact on the semantics of OntoVQL resides in its "connectivity syntax". This visual syntax can be expressed in terms of rules adopting the same form as a production rule in a formal grammar. Thus, similar to what a typical formal grammar is composed of, our grammar has a finite set of non-terminal symbols: `<Query>, <ROLE>, <AND GROUP>, <OR GROUP>, <AND NODE>, <OR NODE>, <entity>`, a finite set of terminal symbols:`<concept>, <individual>, <unknown concept>, <role>`, each of which has its visual equivalence as shown in Figure 3.18, and a finite set of production rules in XML like syntax that are listed below.

Grammar generating the visual language:

```
(1) <Query> ⇒ <concept> |
              <ROLE> |
              <AND GROUP> |
              <AND NODE> |
              <OR GROUP> |
              <OR NODE>
(2) <ROLE> ⇒ <entity> <role> <entity>
```

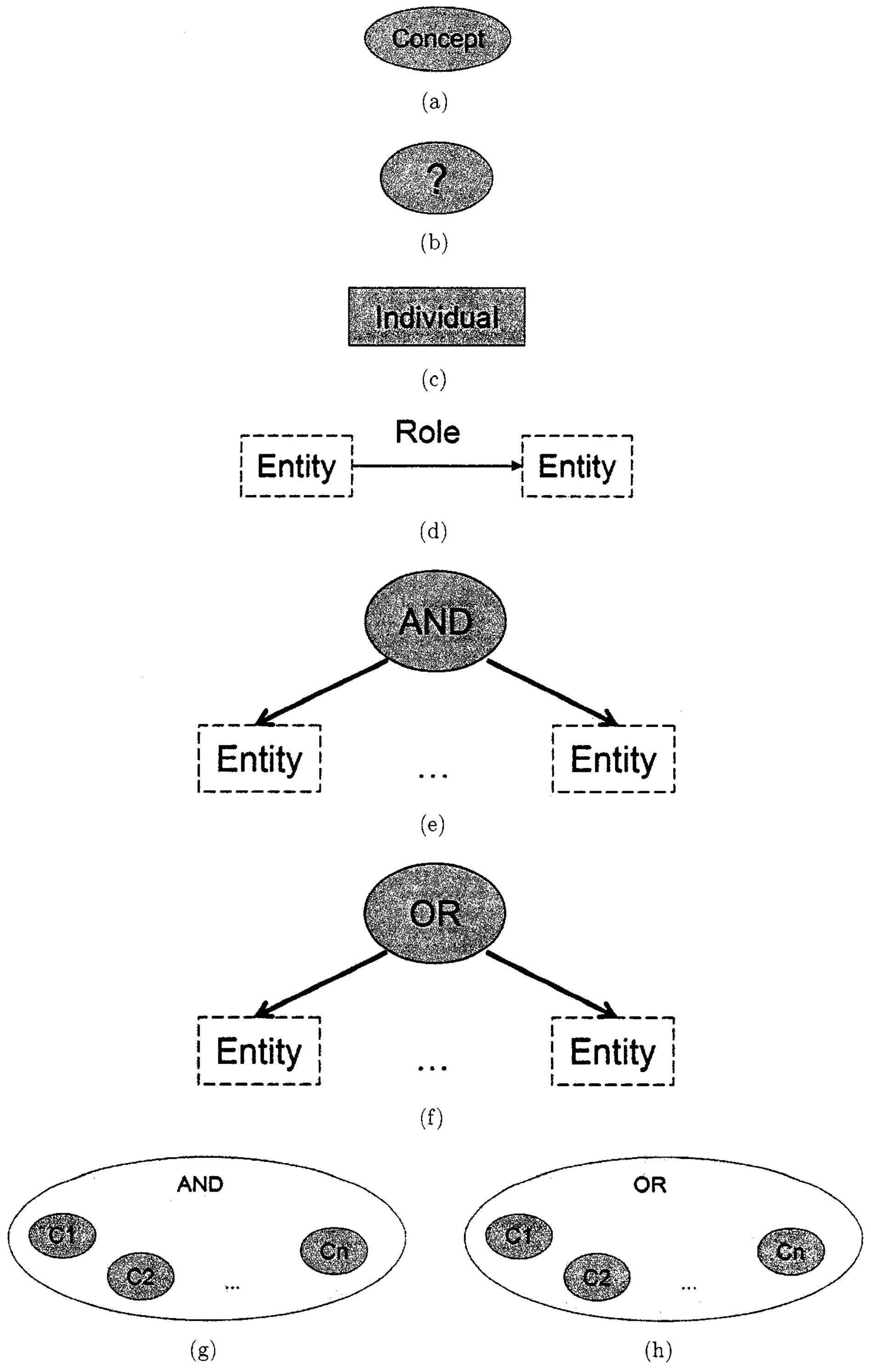


Figure 3.18: (a) <concept> (b) <unknown concept> (c) <individual> (d) <ROLE> (e) <AND NODE> (f) <OR NODE> (g) <AND GROUP> (h) <OR GROUP>

```
(3) <entity> ⇒ <concept> |
               <unknown concept> |
               <individual> |
               <AND GROUP> |
               <OR GROUP> |
               <AND NODE> |
               <OR NODE>
(4) <AND GROUP> ⇒ <and group> <concept>* </and group>
(5) <OR GROUP>  ⇒ <or group> <concept>* </or group>
(6) <AND NODE>  ⇒ <and node>(<OR GROUP>* |
                 <OR NODE>* |
                 (<unknown concept><role><entity>)*)
(7) <OR NODE> ⇒ <or node>(<AND GROUP>* |
                <AND NODE>* |
                (<unknown concept><role><entity>)*)
```

The rules control how the visual entities are allowed to be connected together by directed connection edges and roles. The first rule in the grammar indicates that a query can simply be a concept or consists of more complex components. Among these components, the role is described in the second rule as a binary component where the domain and range are entities that can be expanded to any of the elements listed in the third rule. It is important to mention for the second rule that if a query has the shape of a tree, no role can link any of the query elements under a distinct "branch" of the tree. For example, none of the AND groups in Figure 1 can be linked by a role. The fourth and fifth rules illustrate the AND/OR group containing one or more concepts that are either intersected or unified. Note that an XML like syntax is used for representing the notion of a group by having a start and end tag enclosing one or more concept element. From rule 6, an AND node can be connected to an

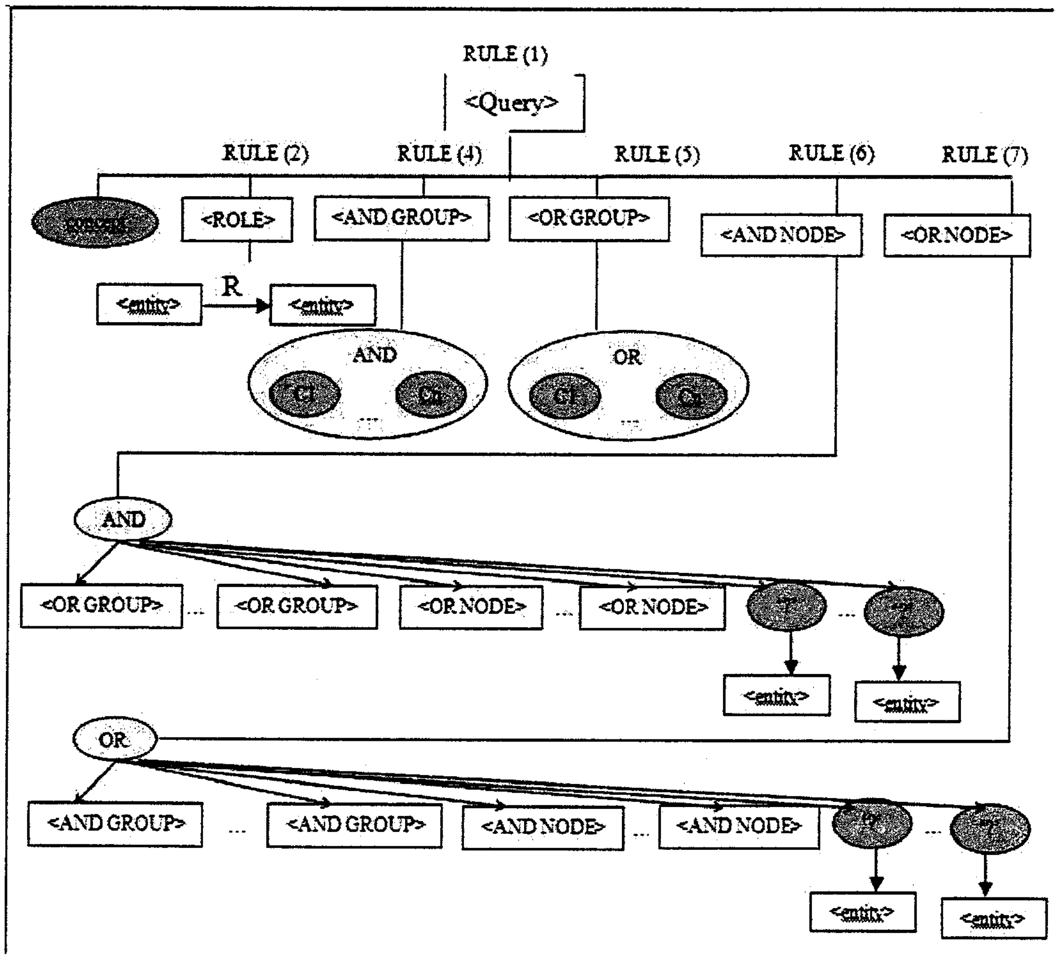


Figure 3.19: Visual description of the grammar rules.

OR group and/or to an OR node and/or to an unknown concept that is linked to a role. A directed connection edge whose source is the AND/OR node and target is the AND/OR group or the unknown concept links the domain and range entities together. For more clarification of the above rules, Figure 3.19 gives a visual description of them as to how the graph query components are linked together in every rule.

As an illustrative example of how the above grammar describes the visual syntax of OntoVQL, consider the query in Figure 3.20. The query is composed of an OR node that links an AND group with an unknown concept related to another unknown concept by a role R1. Note that the query's visual syntax must follow the 7th rule

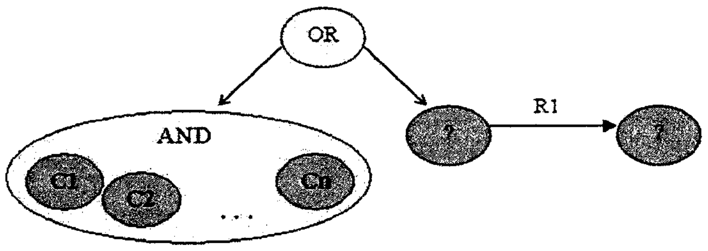


Figure 3.20: An example query generated by the 7th rule of the grammar.

from the grammar above because it contains the OR node.

3.5 Mapping of OntoVQL to the nRQL language

The expressive power of OntoVQL can be informally described as being able to formulate queries on DL Abox elements (concepts and role assertions) and make use of conventional operators (union, intersection) for building up more complex, refined queries. Our proposed VQL is designed independently of any OWL query language and offers basic functionalities for querying OWL-DL ontologies. This means that there is not a one-to-one mapping between the visual components of OntoVQL and the elements of an OWL query language. Note that a number of these elements have no visual counterpart and therefore, OntoVQL does not match the full expressive power of OWL query languages. It follows that OntoVQL is unquestionably less expressive than an OWL query language. However, we claim that for the simple queries mostly asked by the naïve user, and also for complex queries containing conjunction and union, the OntoVQL version of the query exhibits a lower complexity by mainly getting rid of the textual syntax and hiding the query variables of the OWL query. In that sense, we can rightfully claim that OntoVQL is less complex than an OWL query language, or more precisely, a subset of it. Also, since there is no specific language to which OntoVQL must be mapped to, there is the possibility of translating the visually expressed query semantics into a written version using the query language of your choice. In our case, the translation is done using nRQL. However, as has been illustrated in previous sections, the design of OntoVQL was strongly guided by the functionality of nRQL.

The visual mapping of OntoVQL into nRQL consists of translating the semantics of the visual components and their combinations into nRQL expressions. Starting with the concept node, the most basic visual component in OntoVQL, its equivalence in nRQL corresponds to the concept query atom (Figure 3.21). The second basic query construct in OntoVQL corresponds to the role linking two entities. Its translation into nRQL matches the role query atom explained earlier in Section 3.1.3. As depicted in Figure 3.22, each entity is mapped to a distinct variable standing for the domain and range of the role. The domain and range entities can either be a concept, unknown concept, AND/OR group, or AND/OR node. If the domain and range are unknown concepts, then there is no need to include anything but the role query atom in the query body. Otherwise, the body should comprise of the domain and range declarations consisting of the appropriate nRQL translation. Note that although no AND operator is present in the visual query, all the elements in the corresponding nRQL query are intersected. This is an implicit conjunction in contrast to the explicit one that is visualized by an AND node/group. Therefore, outgoing roles from an entity as well as domain and range specification for a role require the use of conjunction in nRQL. Beside the concept and role constructs, there are the union and conjunction operators. As seen earlier, the basic constructs of that type of operations are the AND/OR groups. They are simply mapped to a query body containing the 'and'/'or' keyword followed by a list of concept queries that correspond to the concepts inside the group (see Figure 3.23). Finally, the AND/OR node is mapped similarly to the AND/OR groups with the exception that instead of a list of only concept queries following the 'and'/'or' keyword, there can be a list of all kinds of queries (Figure 3.24). Note that when mapping to nRQL, each AND/OR operator is mapped to a single variable.

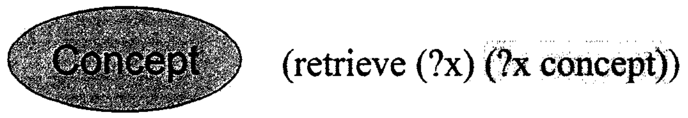


Figure 3.21: Concept query mapping.

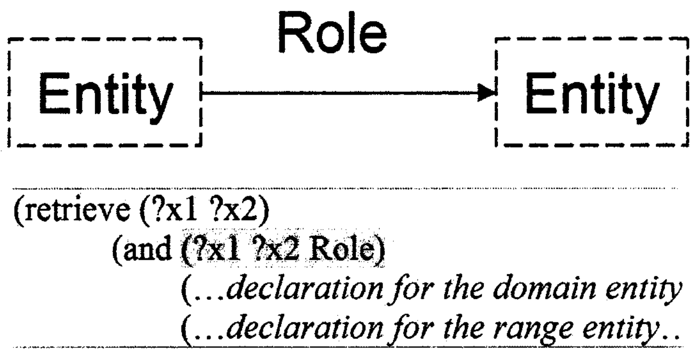
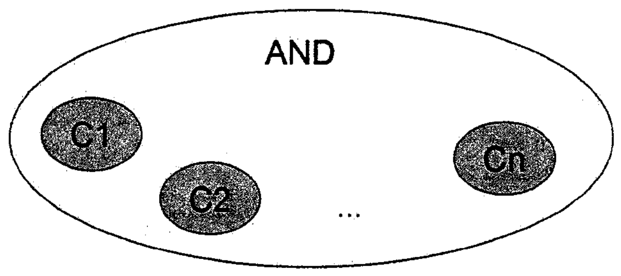

Role
Entity
Entity
(retrieve (?x1 ?x2)
(and (?x1 ?x2 Role)
(...declaration for the domain entity
(...declaration for the range entity..

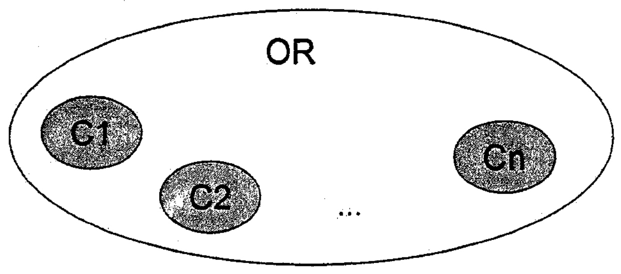
Figure 3.22: Role query mapping.

In order to get an idea of how the mapping is done with more than one construct in the query, consider the example of a graph query with a role and an OR group as shown in Figure 3.25. When mapping to nRQL, each entity must be mapped to a unique variable. In the case of the OR group, the concepts inside it must be mapped to the same variable ?x3. Therefore, in the nRQL translation, the concepts inside the OR group are both mapped to x1 whereas the animal concepts are mapped to different variables x2 and x3. The reason why concepts involved in an AND/OR operation must be mapped to the same variable is because this has a direct impact on the result. For example if concepts C3 and C2 were intersected, then the nRQL query body will include (and (x1 C3) (x1 C2)) which results in binding the variable x1 to those individuals which are instances of concepts C3 and C2. If distinct variables were used instead, then the answer would be those individuals which are bound to C3, plus those individuals which are bound to C2 but not already mentioned for C3. Hence, the semantics changes into adding up C2 and C3 individuals instead of intersecting



```
(retrieve (?x)
          (and (?x C1)
               (?x C2)
                ...
               (?x Cn)))
```

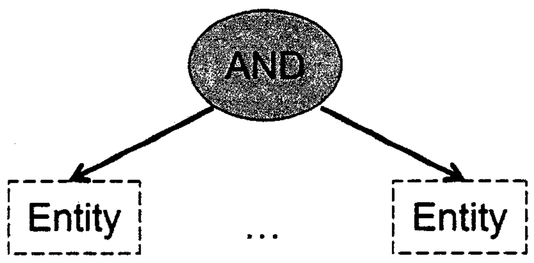
(a)



```
(retrieve (?x)
          (or (?x C1)
              (?x C2)
               ...
              (?x Cn)))
```

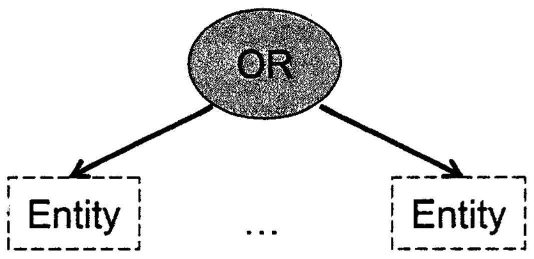
(b)

Figure 3.23: (a)AND group mapping. (b) OR group mapping.



```
(retrieve (?x)
          (and (?x ...)
                  ...
               (?x ...)))
```

(a)



```
(retrieve (?x)
          (or   (?x ...)
                  ...
                (?x ...)))
```

(b)

Figure 3.24: (a) AND node mapping. (b) OR node mapping.

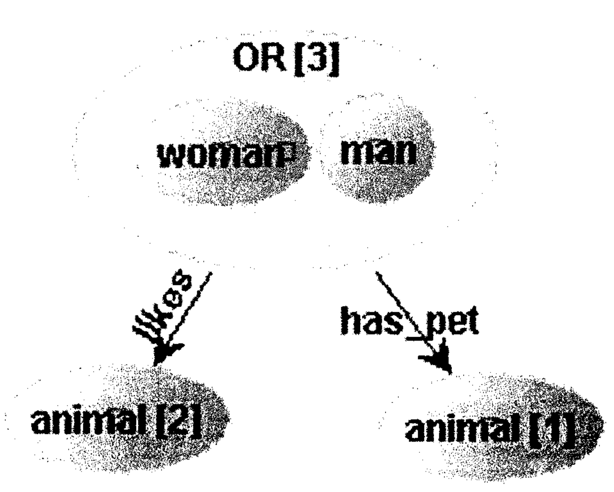


Figure 3.25: Example of a graphical query and its equivalence in nRQL Table 3.17.

```
(retrieve ( $?x2 $?x3 $?x1 )
          (and ($?x2 |animal|)
                ($?x1 |animal|)
                (or ($?x3 |woman|)
                    ($?x3 |man|))
                ($?x3 $?x1 |has-pet|)
                ($?x3 $?x2 |likes|)))
```

Table 3.17: Translation of query in Figure 3.25 to nRQL

them. This is why variable mapping is crucial when translating the visual query into nRQL.

3.6 OntoVQI's characteristics

In order for the visual query language to be utilized, it needs to be part of a visual query system. A prototype has been implemented for this very purpose. This section describes the system's characteristics and properties in terms of what are the services provided and how to use them. In Figure 3.26, the screen shot of the system's main window shows its most important components. The main toolbar provides the basic operations of the system. The first operation to be performed before querying is to load the ontology that is to be queried. The information tabs provide the list of concepts, roles and individuals present in the loaded ontology. It is from those lists

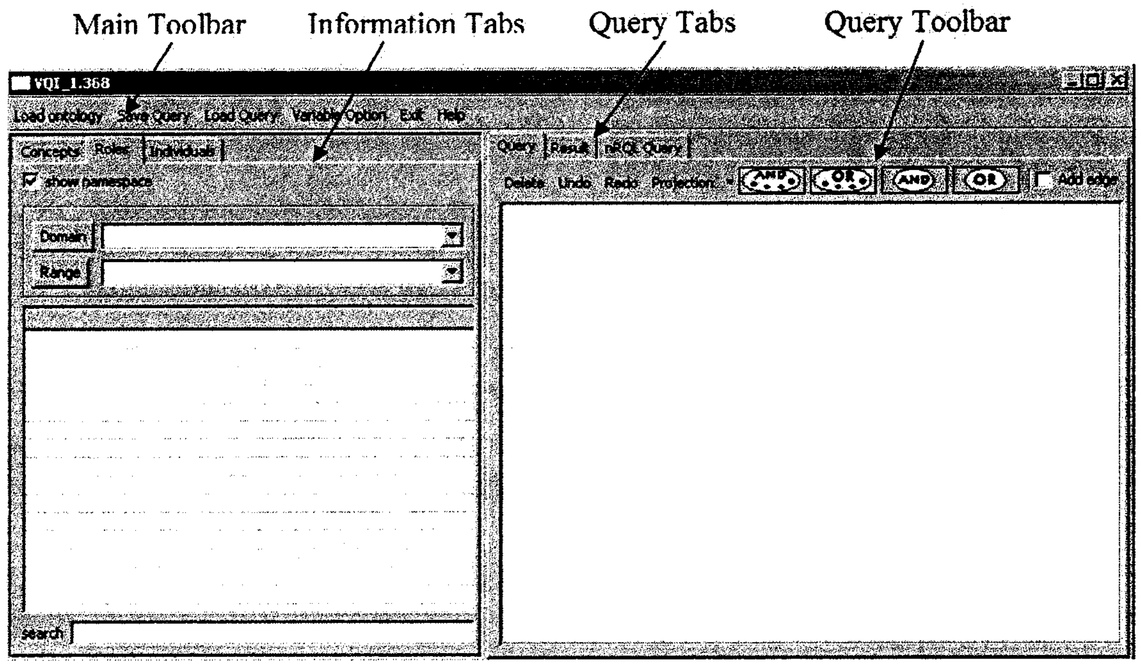


Figure 3.26: OntoVQI's Main window.

that the user will select the parts that will make up the queries to be built. The query tabs consist of the tool's main component since this is where the query pane for query construction is found. Enclosed within the query tab is the query toolbar that provides query constructors along with the basic operations needed for query manipulation. These parts will be explained with greater detail in the subsequent sections.

3.6.1 Main toolbar

The main toolbar provides the global main operations of the system, which are loading an ontology, saving and loading a query and setting the variable option.

Load ontology

No queries can be built if no ontology is loaded into the system. Therefore, the first task to perform after having the system running and functional is to press the "Load ontology" button from the main toolbar. This will open the "Select Ontology" file

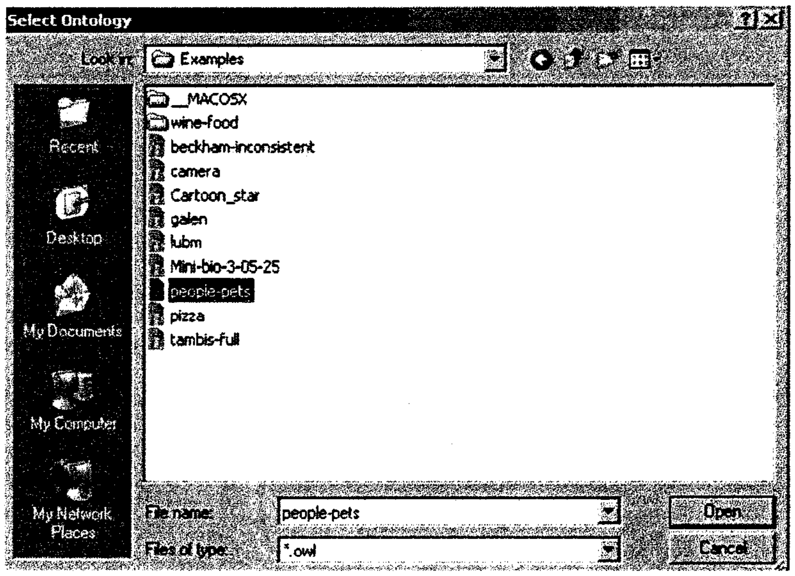
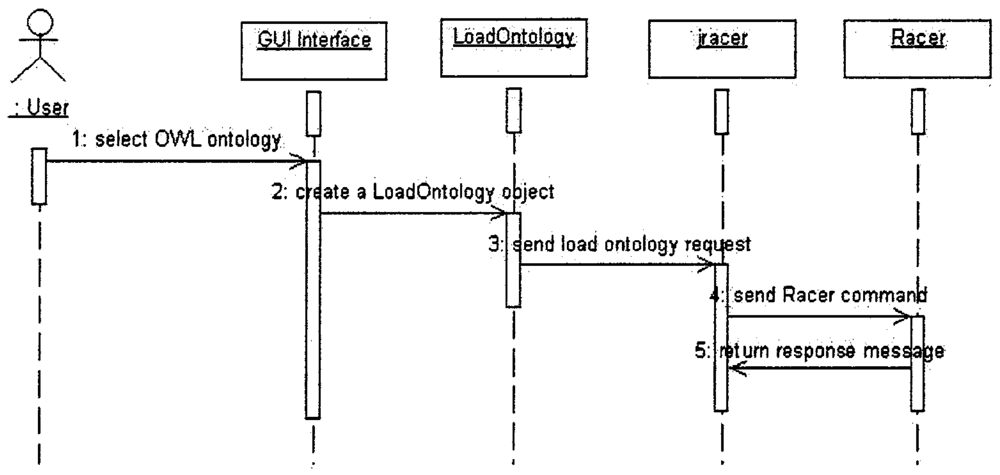


Figure 3.27: File selector for choosing an OWL ontology.

selector (see Figure 3.27) which filters the OWL ontologies which are the files having the .owl extension. Figure 3.28 shows how this operation takes place. First, the user inputs the ontology file location to the OntoVQI tool by selecting an ontology and pressing on the "open" button in the file selector. This information is addressed to the system's "Load ontology" module, which will in turn send the load ontology command to Racer. The reasoner then executes the command by loading the OWL file into its system and sends a success or failure message back to the VQS. Therefore, the loading process is really done by Racer and into Racer while OntoVQI is just the façade through which the operation is accomplished.

Save/Load Query

One of the most useful services provided by the tool is the ability to save and load a query session. In fact, after having spent a certain amount of time on formulating some meaningful queries, it will be a waste of time to lose all the work done once the system is exited. It may also be that the user is still not finished with the query formulation and needs to refine them some time later. In that case it would be very



: User
GUI Interface
LoadOntology
jracer
Racer
1: select OWL ontology
2: create a LoadOntology object
3: send load ontology request
4: send Racer command
5: return response message

Figure 3.28: Sequence diagram of loading an OWL ontology.

useful to save the current session and load it afterward for that purpose. Also, the save/load utility may be seen as a way for users to share their queries between each other. The way this service works is very simple. Once the user wants to save a query session, the "Save query" button from the main toolbar must be pressed which will open the "Save Query" window (see Figure 3.29). The file name must be filled in and the "Save" button pressed for the operation to proceed. The query session is then saved as an XML file to the specified location. Loading a query session is as simple as saving it. The user needs to press the "Load Query" button from the main toolbar and this will open the "Load Query" window (see Figure 3.30). Then, the user can select from the list of XML filtered files the saved query session in question. How the queries are transformed from their graph visualization and encoded into XML is done through the XMLEncoder [40] class, which was exclusively designed for archiving graphs as textual representations of their public properties. Similarly, when loading the query session, which is in the form of an XML file, the transformation is done in the other way around by the XMLDecoder [39] class, which reads the XML data and creates objects according to the content.

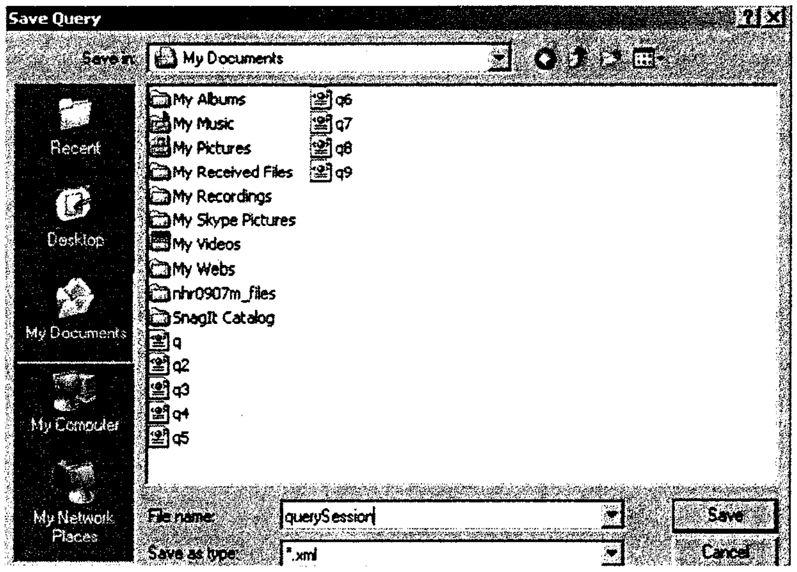


Figure 3.29: Save query window.

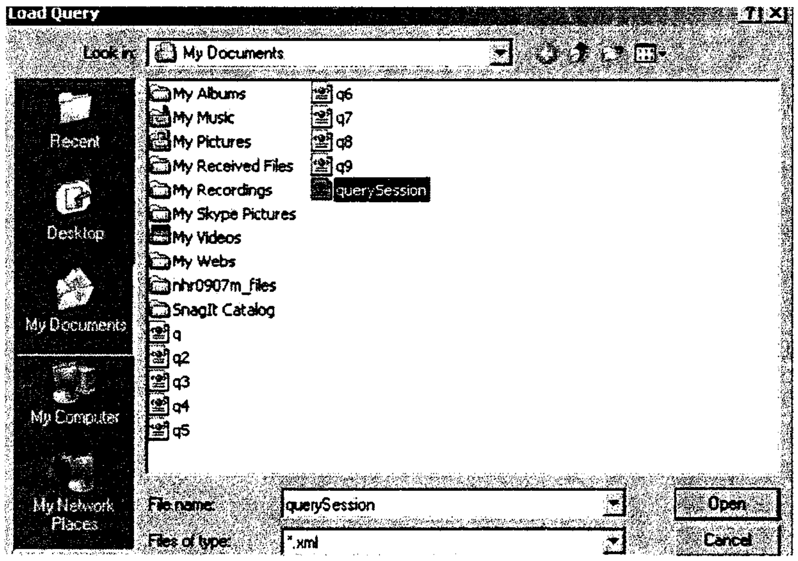
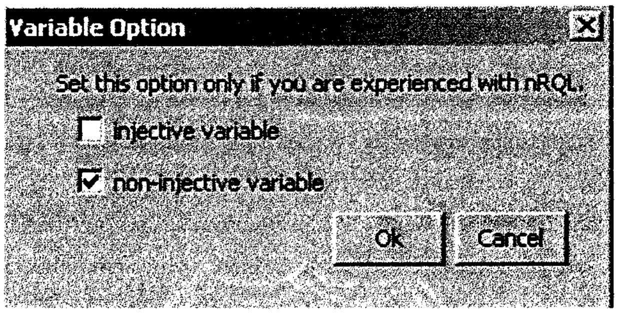


Figure 3.30: Load query window.

Variable option

As stated previously in Section 3.1.3 there are two variable types available for writing a query in nRQL: injective and non-injective variables. As it was explained, the variable choice is important since it can affect the outcome of the queries. Therefore, OntoVQI should provide the option of setting the variable type. This option is more specifically intended to the user who is expert in nRQL since it is more likely for this type of user to make the distinction between these two types of variables and to make a practical use of it. On the other hand, the naïve user, i.e. the one who is not familiar with nRQL, can obviously use this option to explore OntoVQI services but it should be mentioned that it is not particularly recommended for this type of user to depend on it. The reason for that relies on the possibility for this option to be a source of confusion to this user considering that a given query can have different outcomes as shown in the example in Section 3.1.3. This option is accessible by clicking on the "Variable Option" button of the main toolbar. This will open the "Variable Option" window (Figure 3.31), which consists of two checkboxes that can be exclusively checked for either the injective variable option or the non-injective variable option. Note that by default, variables are set to be non-injective. This is the case because the non-injective variable option permits to obtain all the tuples in the result that are to be found because of its binding principle explained earlier in the same Section of 3.1.3. On the other hand, even though the injective variable option allows the elimination of the redundant results that would be otherwise obtained in the case of having non-injective variables instead, it may be more desirable for the naïve use to have the complete result than to have a redundant-free one. If the user chooses the non-default option, then all the subsequent queries will be translated to nRQL with variables being injective. However, those queries that have been translated prior to this change will remain the same. For example, the two queries in Figure 3.32 are apparently the same; nonetheless, the result cardinality is not the same. The first


Variable Option

Set this option only if you are experienced with nRQL.

[ ] injective variable

[✓] non-injective variable

Figure 3.31: Variable option window.

query, Query 5, has been translated with non-injective variables, which was before the variable setting got changed to injective variables. The second query, Query 10, was added after that change and thus its translation into nRQL was with injective variables. The outcome of the second query with no tuples in the result and the first one with two can be explained by examining Figure 3.33. The latter shows that the tuples in the result of Query 5 have the second and third argument filled with the same cat instance, meaning that there was no person who had more than one cat. Therefore, it is expected to have zero tuples with the injective variable option since this would involve finding a distinct cat instance to bind for each cat variable. As a result, two queries that are visually identical and are present conjointly in the query pane are displaying different results. This situation can be seen as an example of why it is desirable to offer the variable setting service but that it could become puzzling at the same time especially for the naïve user.

3.6.2 Information tabs

The information tabs consist of an essential component in OntoVQI. It is called as such because it is the part of the tool that provides the list of concepts, roles, and individuals as the information about the ontology that is needed to formulate queries. In order to facilitate the procedure of finding the desired information from these lists, a case sensitive search field is available in each tab to perform a search over the

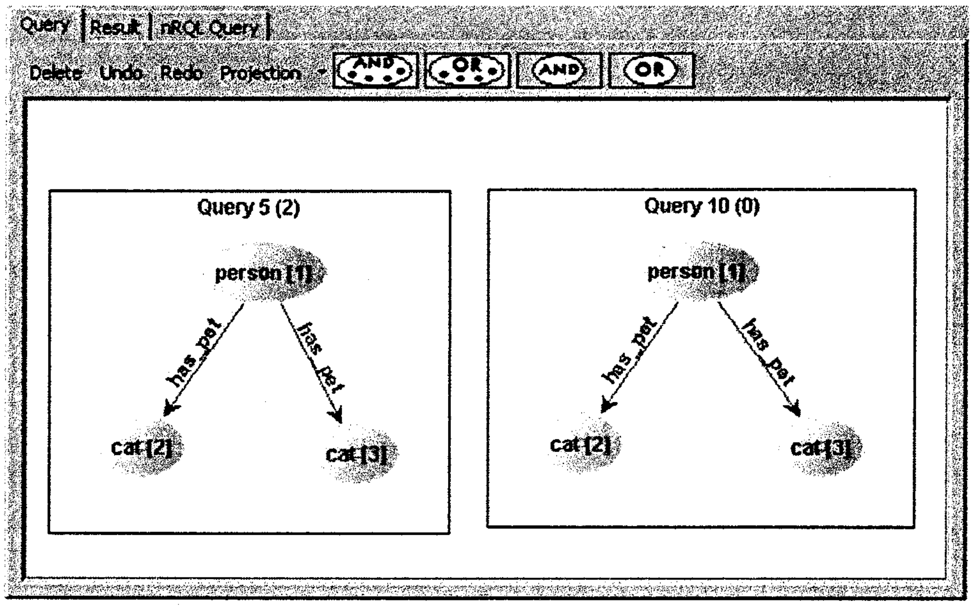
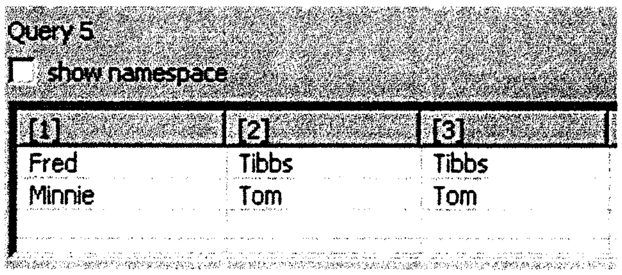


Figure 3.32: Variable type effect on query result.


Query 5

show namespace

[1]	[2]	[3]
Fred	Tibbs	Tibbs
Minnie	Tom	Tom

Figure 3.33: Result for Query 5 in Figure 3.32.

alphabetically ordered list of names. Note that the list of names refers to the concept/role/individual names excluding the namespaces. Therefore, the search is done over the names without namespaces even though the list is composed of namespace plus name elements. By default, these lists include the namespace alongside the name but they can be simplified into a list of names by unselecting the "show namespace" checkbox situated at the top of each tab. The retrieval of the lists of concepts, roles and individuals is conducted immediately after the ontology to be queried is loaded. For the obtainment of each list, first, the appropriate command is sent to Racer, second, the returned result is parsed and processed into a list of namespaces and a list of names, third, the list of names is alphabetically ordered, and finally, the lists are fetched into their corresponding table.

Concept tab

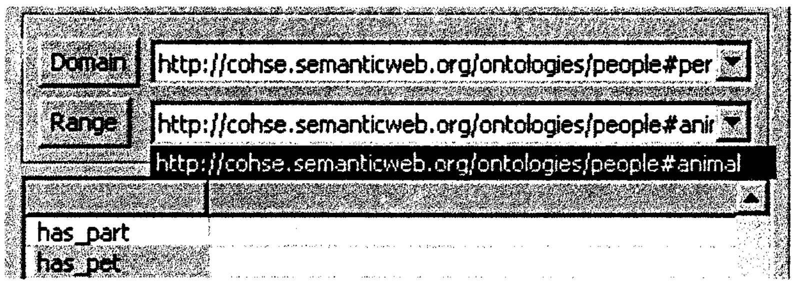
The concept tab mainly consists of the list of concepts present in the loaded ontology. The service provided on this tab is to find the parent/child of a given concept. An ontology can be described as a taxonomy of terms whose root is the Top concept that encompasses all concepts and whose leaves is the Bottom concept that is encompassed by all concepts. Therefore, each term in the taxonomy except for the Top one is derived or more precisely subsumed by one or more of these terms and is itself deriving or subsuming other terms.

Role tab

In the role tab, beside the search field, there is the possibility to find the domain and/or range concept(s) of a given role or object property. In fact, an OWL ontology may contain the specification of the domain and range of a role. Thus, the "has-pet" property OWL definition below taken from the people-pet.owl shows that it has a domain of "person" and a range of "animal", meaning that it relates instances of the class "person" to instances of the class "animal". Therefore, this is a way of

```
<owl:ObjectProperty rdf:about"#has-pet">
        ...
        <rdfs:domain>
        <owl:Class rdf:about"#person"/>
        </rdfs:domain>
        <rdfs:range>
        <owl:Class rdf:about"#animal"/>
        </rdfs:range>
</owl:ObjectProperty>
```

Table 3.18: Domain and range specification of a role in OWL.



http://cohse.semanticweb.org/ontologies/people#per
http://cohse.semanticweb.org/ontologies/people#anir
http://cohse.semanticweb.org/ontologies/people#animal

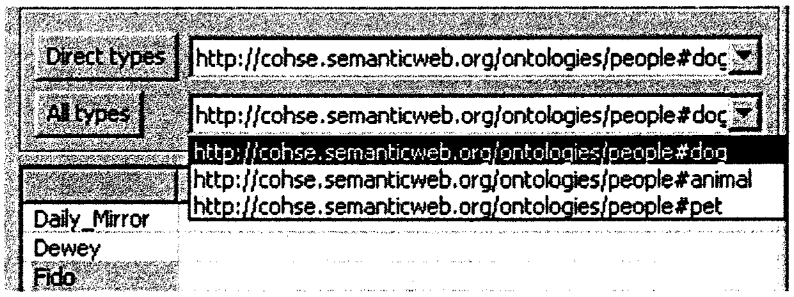
Figure 3.34: Domain/range of a selected role.

finding out how the roles are connecting instances of concepts to each other and hence permitting the user to get an idea of how the ontology describes the knowledge domain and what are the concepts that most likely will return a non zero result when queried.

In order to obtain the domain/range concept(s) of a role, the user must select the role from the table in the Role tab. In Figure 3.34, it is the "has-pet" property that has been selected. Then the domain/range buttons must be pressed before the drop down lists of the domain and range concept(s) of the selected role appear. This operation is carried out in the same manner depicted in Figure 3.31 in Section 3.6.

Individual tab

The individual tab contains the service of finding the direct type and/or all types of a given individual. The direct type is simply the most specific atomic concept of which an individual is an instance. On the other hand, the "all types" option



http://cohse.semanticweb.org/ontologies/people#doc
http://cohse.semanticweb.org/ontologies/people#doc
http://cohse.semanticweb.org/ontologies/people#dog
http://cohse.semanticweb.org/ontologies/people#animal
http://cohse.semanticweb.org/ontologies/people#pet

Figure 3.35: Direct type/all types of a selected individual.

serves to retrieve all atomic concepts of which the individual is an instance. This service is useful in helping the user finding out which concepts are to be queried with meaningful results. Also, it can be of use in determining the types of the individuals returned in the result of a query. As shown in Figure 3.35, by selecting an individual from the table in the individual tab and pressing on the "Direct types" or "All types" button, a drop down list of the corresponding concepts appears.

Drag and drop

An important usability feature in the system that helps to provide a richer user interface is the drag and drop feature. In the information tabs, the elements in the tables can be dragged and dropped into the query pan^. In Figure 3.36, each query represents a dragged and dropped element from every table. For the first query, the dropped cat concept stands on its own as a query formed of a single concept whose result cardinality reveals that there are two cat instances in the ontology. In the case of the second query, the "has-pet" object property name was dropped into the query pane and the query formed consists of a role whose domain and range are unknown concepts because the intended meaning of this query should stand for finding the pairs of individuals related by the "has-pet" role without any restriction on the individual's type. On the other hand, unlike the two previous dropped elements, the third dropped element, which is an individual, does not stand for a query. This is because an individual on its own does not represent any query semantics. Hence, there is no

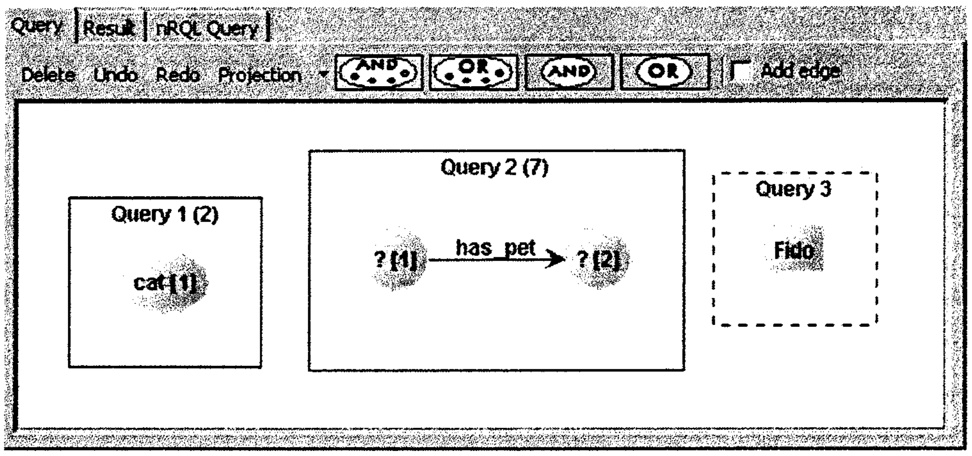


Figure 3.36: A dragged and dropped concept/role/individual.

cardinality for the result as for the other queries. However, this dropped individual can be linked to other queries with some role. These three queries characterize the three basic building blocks that can be dragged and dropped from the information tabs. The concept and the role queries are the ones to start with, as they are the simplest queries that can be formulated with the tool.

The "Add edge" option

The "Add edge" option is provided in order to be able to connect an 'AND'/'OR' node to an 'AND'/'OR' group. By default, users can connect query components with a role which is represented by a filled arrow. However, users wishing to link query components for intersection or union ('AND'/'OR' nodes to 'AND'/'OR' groups), they must use a different kind of arrow which is enabled by checking the "Add edge" checkbox in the query toolbar (Figure 3.40).

3.6.3 Query tabs

The agglomeration of the query tabs is the main component in the OntoVQI tool. This is where queries are constructed, their results can be consulted, and their translation into nRQL can be examined.

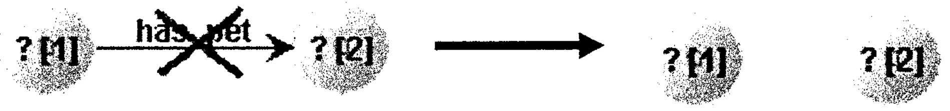


Figure 3.37: Deleting a query component inconsistently.

Query tab

The query tab is composed of the query pane, in which the query construction takes place, and of the query toolbar providing the services essential for assembling the simplest queries into more complex ones. The query toolbar also provides access to operations having a direct effect on the queries: delete, undo/redo and projection.

Delete

With every modification performed on a query, it is instantly reevaluated for retrieving the new result cardinality. Therefore, no matter how a query is manipulated, it must always be in a consistent state meaning that it should always follow OntoVQL syntax rules described in section 3.4 to be in a state ready for evaluation. When deleting the component of a query, the outcome may be inconsistent. In fact, the deletion of the "has-pet" role in Figure 3.37 resulted in the creation of two queries, each composed of the unknown concept alone, which is against the visual grammar rules, which makes these two queries not consistent and it is not possible to evaluate them for results. In order to solve this problem, the delete operation must remove as many query components as necessary for insuring that all the new queries resulting from this deletion are consistent. For example, in the case of Figure 3.37, the deletion of the role would have as consequence the removal of the whole query. Note that the user can delete a query in its entirety by selecting the query box before hitting the delete button.

Undo/Redo

The undo/redo service is one of the most common usability features found in user interfaces. Whether the user is a novice learning to use the tool or an expert who just hit the wrong key, the application should provide the ability to quickly undo the results of their actions. Therefore, undo/redo is an essential service for improving OntoVQI's usability. What would be expected from the undo feature is to undo the list of actions performed on the graph query. However, in OntoVQI, only those actions that have an impact on the query semantics can be undone as deleting or adding a query component. This means that the displacement of queries or query components is not registered in the list of actions to be undone since this action has no effect on the query semantics.

Query preview

The query preview is provided as an instant feedback to the user in order to help him constructing meaningful queries. It consists of the number of tuples that are found in the result returned by Racer. With every addition, deletion, or modification made to a query, its preview gets instantly evaluated by sending the preview command to Racer. As an example demonstrating the usefulness of this feature, the user usually starts a query construction with a simple concept or role query. If the preview for this simple query is found to be zero, then there would be no point in going any further with this query since there would be no meaning to construct a query for which we know in advance it will have no result. Therefore, in that sense, the query preview allows the user to save time by instantly providing the cardinality of the result.

Projection

The projection service refers to what was explained previously about the head of a query in nRQL, i.e. the head consists of all or some of the variables appearing in the query body and on which the result is projected. In OntoVQI, projection provides the

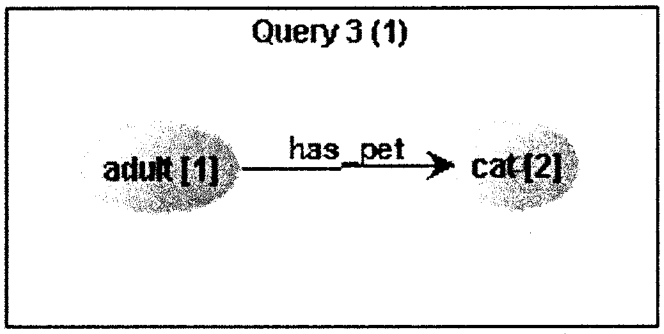
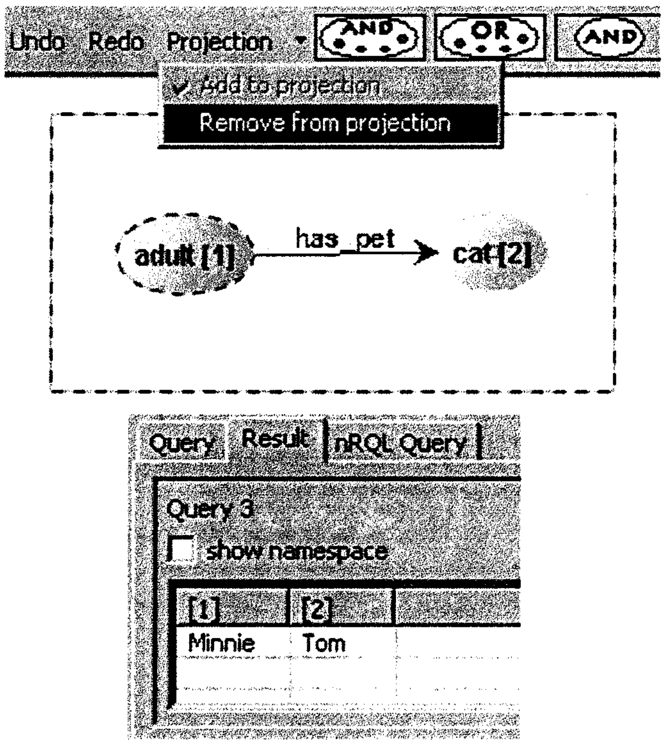


Figure 3.38: A simple role query.

ability of selecting which parts of the graph query or equivalently which variables of the nRQL query should be projected on in the result. The number between brackets next to the node's name is an indicator that the node in question has an underlying variable in its nRQL translation as shown in the simple query of Figure 3.38. The difference between a projected and a non-projected variable resides in the color of this number: red when selected as part of the projection, black when it is removed from the projection. By default, these tags are initially included in the projection. Therefore, in order to take out a node from the projection, the user must first of all select the node in question. Then, by clicking on the drop-down list of the projection button, select the "Remove from projection" option, which will cause the tag to its change color from red to black, as it is the case with the adult concept in Figure 3.39 (b). Note that the change in projection had a direct impact on the displayed result. In fact, when both concepts are included in the projection in Figure 3.39 (a), then the result comprises both tags, whereas when only the domain concept of the role query is projected, then the result contains only the second tag. For the purpose of getting the projection back in the node, the user must again select the node and choose the "Add to projection" option from the projection's drop down list.

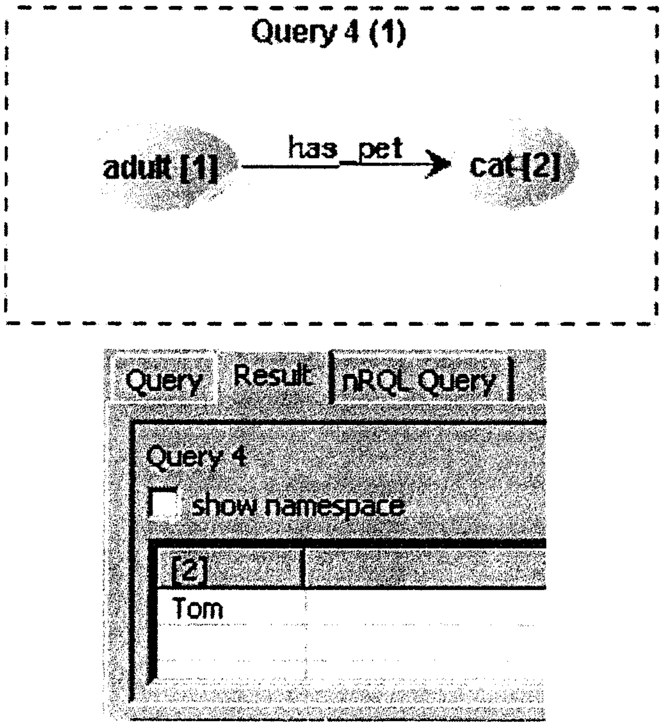


Query 3

show namespace

[1]	[2]
Minnie	Tom

(a)



Query 4

show namespace

[2]
Tom

(b)

Figure 3.39: Removing a node from the projection.

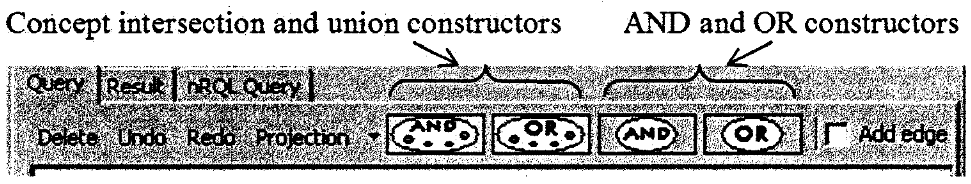


Figure 3.40: The intersection and union constructors in the query toolbar.

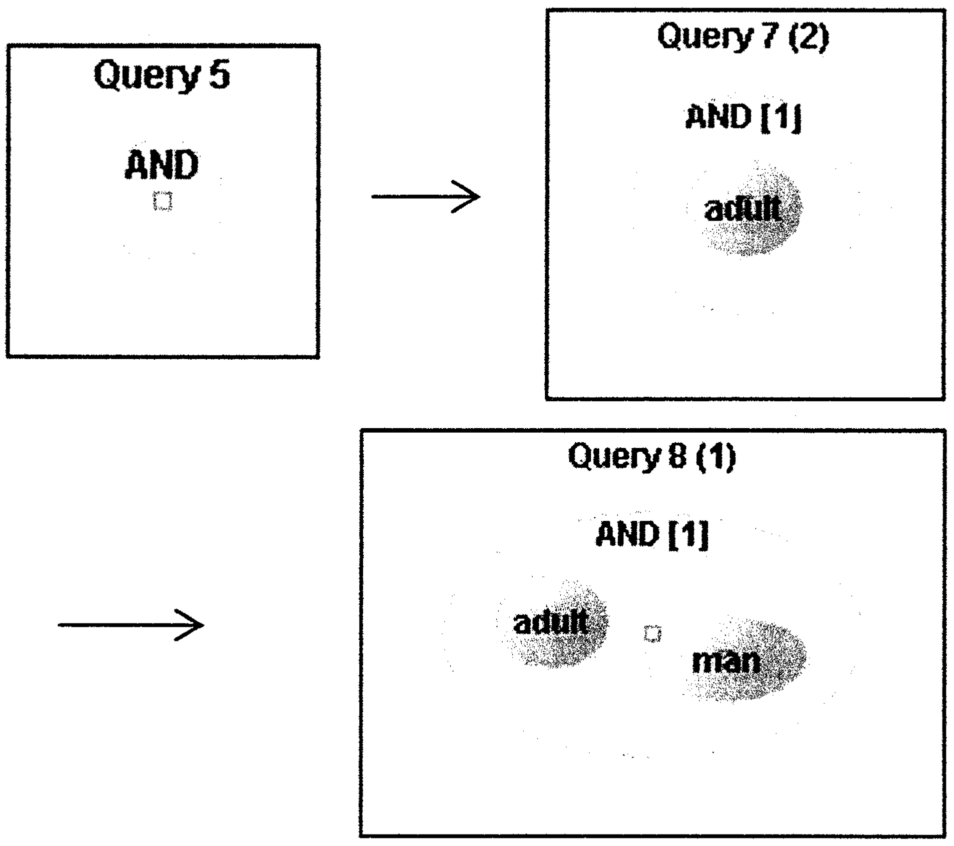
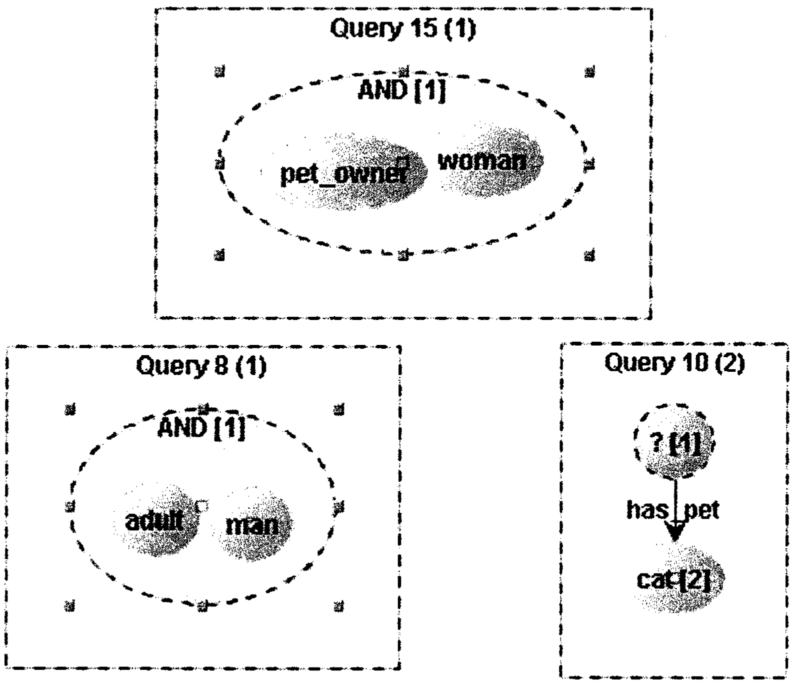
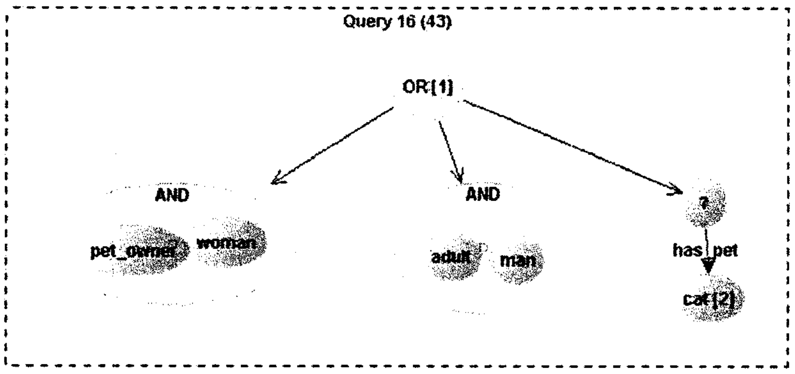


Figure 3.41: The use of the concept intersection constructor.



(a)



(b)

Figure 3.42: Using the OR constructor.

Intersection and union constructors

As it was previously mentioned, there are two types of intersection and union constructors. The first one, concept intersection and union constructors, serves to group concepts whereas the second one, the AND and OR constructors, mainly serves to group these groups of concepts. Icons in the query toolbar as shown in Figure 3.40 depict these constructors.

In order to use the concept intersection constructor, first the user needs to drag and drop the icon that stands for it into the query pane. Figure 3.41 shows that this results in the creation of a non-evaluated query containing an empty "AND group". Then, concepts can be added one by one into the group by being dragged either from the concepts table or the query pane and dropped on top of it. Similarly, the concepts union constructor is used in the same way. As stated previously in the visual grammar rules, it is possible to perform the union operation on AND groups and unknown concepts related to a role by the means of the OR constructor. First the AND groups and unknown concepts to be unified must be selected (Figure 3.42 (a)). Then, the icon representing the OR constructor must be dragged and dropped in an empty space of the query panel. An OR node will appear at the same position where the icon was dropped and directed edges would be drawn from the OR node towards the selected items (Figure 3.42 (b)). The same applies for the AND constructor except that instead of selecting AND groups, it should be OR groups that have to be selected.

3.7 System Design and Architecture

The design approach followed in implementing OntoVQI is based on the architecture of the three primary layers [17]: presentation, domain and data source. The following explains in general the role of each layer and what it is about.

- The presentation layer is about how to handle the interaction between the user and the software. This can be a command-line or text-based menu system, but most likely it is a rich-client graphics User Interface or an HTML-based browser User Interface. The primary responsibilities of the presentation layer are to display the information to the user and to interpret commands from the user into actions upon the domain and data source.

- The second layer is the domain logic which also refers to the business logic. This is the work that the application needs to do for the domain we are working with. It involves calculations based on input and stored data, validation of any data that comes in from the presentation layer, and figuring out what data source logic to dispatch, depending on commands from the presentation.
- The last layer is the data source which is about communicating with other systems that carry out tasks on behalf of the application. These can be transaction monitors, other applications, and so forth. For most applications, the biggest piece of data source logic is a database that is primarily responsible for storing persistent data.

The architecture of our design consists of these three layers. Each layer is composed of one or two packages whose functionality revolves around a specific role (see Figure 3.43). The following subsections covers the Whereabouts of each of these layers and their packages.

3.7.1 Presentation Layer

The presentation layer consists of a rich-client User Interface. Figure 3.43 shows that the presentation layer contains two packages: the graph package and the User Interface package. The User Interface package is responsible for generating a graphical user interface in order to display information and to receive user commands whereas the graph package is responsible for the query graphical visualization. Figure 3.44 shows that the class diagram of the presentation.graph package.

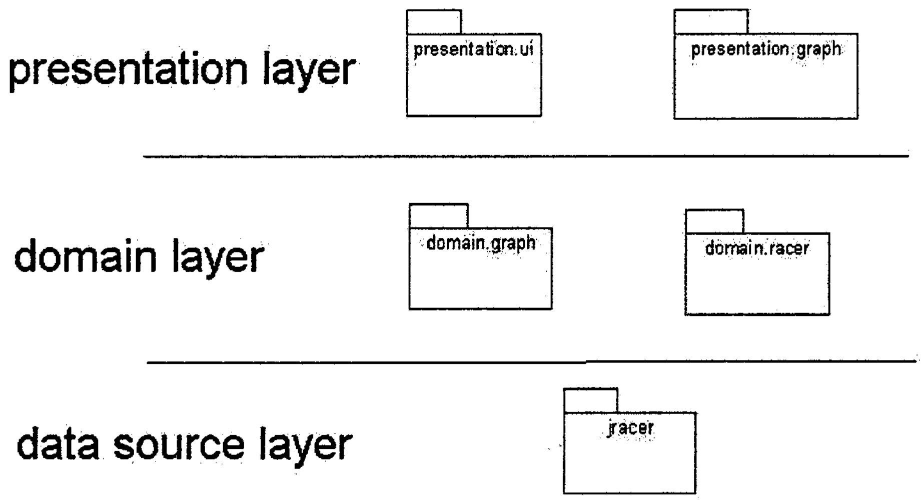


Figure 3.43: Distribution of packages into the three layers: presentation, domain and data source.

The SWT [37], which stands for Standard Widget Toolkit, consists of the third party with which the user interface is implemented. The SWT is an open source widget toolkit for java designed to provide efficient, portable access to the user-interface facilities of the operating systems on which it is implemented. It is analogous to AWT/Swing in Java with one difference - SWT uses a rich set of native widgets.

The graph package is responsible for the graphical display of the query. The graphical components are based on JGraph [19] which is a mature, feature-rich open source graph visualization library written in Java. All the classes in the graph package are extended from the jgraph classes in order to configure the graph according to our needs. The GraphicalQuery class plays a key role as it encompasses the graphical model for all queries and is responsible for getting User Interface requests, like add/remove/modify query components, and forwarding them to the domain layer, but more precisely as it will be seen next, to the domain.graph package.

Furthermore, another third party is used for rendering the graphical query: JGraph Layout Pro. The latter takes graph structures defined using the JGraph library and performs either or both of two specific functions on that graph structure:

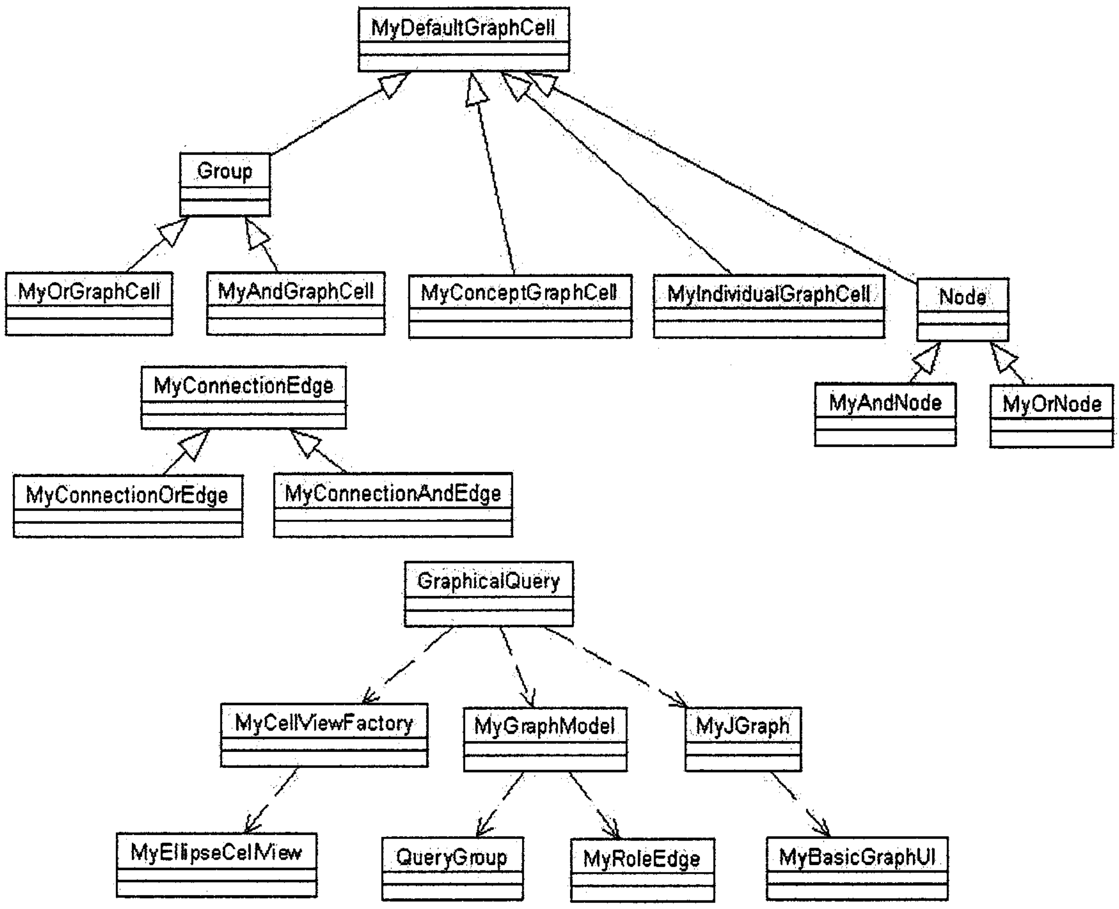


Figure 3.44: Class diagram of the presentation.graph package.

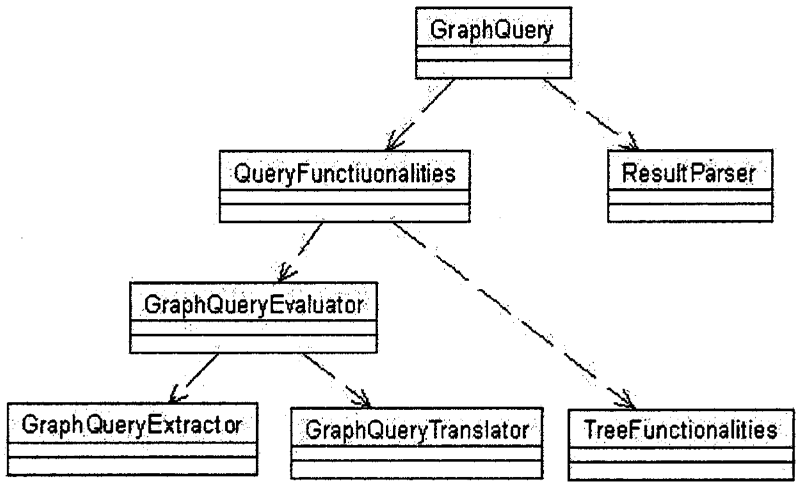


Figure 3.45: Class diagram of the domain.graph package.

- Position the vertices of that graph using an algorithm(s) that attempts to fulfill certain aesthetic requirements.
- Add and remove control points of edges in the graph using an algorithm(s) that attempts to fulfill certain aesthetic requirements.

3.7.2 Domain Layer

As previously mentioned, the requests from the presentation layer are forwarded to the domain layer. In fact, the domain.graph package takes care of these requests by dispatching the appropriate services available through the domain.racer package for query results. For example, if a role is added between two concepts within a query, then, a request is sent to the domain.graph package for adding this new component. The domain.graph package carries out the necessary modifications into the data structures to account for that change and then needs to call racer services through the domain.racer package to update the results of that query and finally, communicates these changes to the presentation layer.

The most important package in the design and implementation of the tool is the

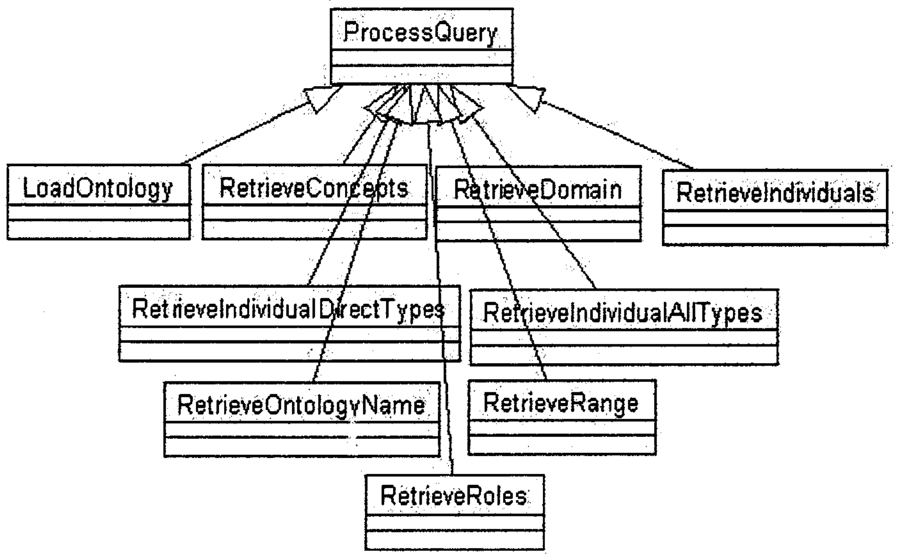


Figure 3.46: Class diagram of the domain.racer package.

domain.graph package since it contains the core of the application's business logic. Each class in the domain.graph (see Figure 3.45) package plays a key role. First, the GraphQuery class corresponds to the data structure of the graphical query. Each instantiated graphical query has its corresponding GraphQuery object associated with it. It contains the API for setting/getting the nRQL translation, the query result and the graph query components (ex.: concepts) and features (ex.: query preview). The essential task of the GraphQueryExtractor class is to extract each indivual graph query from the graph model components and then assign each of these graph queries to the corresponding GraphQuery object. The GraphQueryTranslator class takes the query data structure, that is the GraphQuery object, and performs the mapping of the internal graph structure into nRQL. The ResultParser class is given the query result from Racer to parse and save within a data structure for the presentation layer to display. The whole process of extracting, translating, retrieving and processing query results is orchestrated by the GraphQueryEvaluator class. The domain.racer package is responsible for processing the nRQL query and sending it to Racer (see

Figure 3.46).

3.7.3 Data Source Layer

The essential and only objective of the data source layer in the application is to communicate with RacerPro for submitting nRQL queries and obtaining their results. The source package makes use of the JRacer library which consists of a Java-based API for RacerPro to call its services. In fact, the source package contains only one class, the racer class, whose functionality is to set/get Racer commands and Racer results.

Chapter 4

Conclusion

Applying visual techniques for accessing data has been a subject of research in the database domain that has been given attention and effort with the multiple visual query systems and visual query languages proposed. The trend continued with the effort of visually representing knowledge in an ontology. However, not much has been done for visually querying it. In that perspective, the main goal of this research was to propose a formal visual query language for querying an ontology. OntoVQL represents the realization of that goal. Even though OntoVQL was specifically mapped to nRQL, it was initially designed to be independent of any particular querying language for ontologies and thus has the basics to be easily adapted and mapped to any of these languages independently. As for the expressivity of OntoVQL, its union and intersection constructors allow to build more than simple queries. OntoVQI is the prototype that has been implemented to effectively build queries in OntoVQL and view their results. OntoVQI is a visual interface that has been developed with features, such as query preview, that are meant to help the user towards building meaningful queries.

There exists no proper or formal way to measure the performance or effectiveness of a visual query language. Hence, the quality of the design of such a language remains theoretical and is open to discussion. In our case, OntoVQI brings the designed visual query language a step further by transposing it from mere theory to an actual implementation that allows the user to have a practical experience with querying an ontology using OntoVQL. Furthermore, given that OntoVQI is in effect a User Interface, it would have been helpful to measure the effectiveness of OntoVQL by conducting a user interface evaluation on OntoVQI. Given that conducting such an evaluation represents by itself a subject of research and due to time restrictions, this evaluation has not been conducted. Therefore, this evaluation can be considered as one major future work contribution in order to get feedback and improve the language and the user interface features of the system.

Before OntoVQL came to its actual form, it was known and previously presented as "GLOO: A Graphical Query Language for OWL Ontologies" [15]. Later, after some modifications were made to the visual query language in order to improve its visual representation of queries and to allow a wider range of queries to be represented, it became known as OntoVQL [16]. OntoVQI has been an inspiration to other people's work as it can be verified with the prototype presented in [5] which is greatly influenced by OntoVQI. Overall, OntoVQL has been cited in a number of articles as in [34], [35] and [30], and has been perceived as the most recent attempt to support a graphical mode for query formulation in the context of the Semantic Web.

Bibliography

[1] M. Andries. Graph rewrite systems and visual database languages, phd thesis, leiden university, netherlands,. February 1996.

[2] N. Athanasis, V. Christophides, and D. Kotzinos. Generating on the fly queries for the semantic web: The ICS-forth graphical RQL interface (GRQL). In *Proceedings of the 3rd International Semantic Web Conference*, pages 486–501, 2004.

[3] F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider, editors. *Description Logic Handbook.* Cambridge University Press, 2002.

[4] F. Baader, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider. Description logic handbook. In *Cambridge University Press*, 2002.

[5] Christopher JO Baker, Rajaraman Kanagasabai, Wee Tiong Ang, Anitha Veeramani, Hong-Sang Low, and Markus R Wenk. Towards ontology-driven navigation of the lipid bibliosphere. *BMC Bioinformatics*, 9, February 2008.

[6] W.L. Bewley, T.L. Roberts, D. Schroit, and W.L. Verplank. Human factors testing in the design of xerox's 8010 'star' office workstation. In *Proceedings ACM CHI'83 Conference*, pages 72–77, December 1983.

[7] D. Brickley, R.V. Guha, and B. Mcbride. RDF vocabulary description language 1.0: RDF schema. In *W3C Recommendation*, February 2004.

[8] T. Catarci, M. F. Costabile, S. Levialdi, and G. Santucci. A graph-based framework for multiparadigmatic visual access to databases. *IEEE Transactions on Knowledge and Data Engineering*, 8(3):455–475, 1996.

[9] T. Catarci, M.F. Costabile, S. Levialdi, and C. Batini. Visual query systems for databases: A survey. *Technical Report SI/RR-95/17*, 1995.

[10] T. Catarci, Santucci G., and M. Angelaccio. Fundamental graphical primitives for visual query languages. *Information Systems*, 18(2):75–98, March 1993.

[11] T. Catarci, T.D. Mascio, E. Franconi, G. Santucci, and S. Tessaris. An ontology based visual tool for query formulation support. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04)*, 2004.

[12] M.P. Consens and A.O. Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proceedings of 9th ACM SIGA CT-SIGMOD Symposium on Principles of Database Systems*, pages 404–416, 1990.

[13] F. Cruz. Doodle: A visual language for object-oriented databases. *ACM-SIGMOD International Conference on Management of Data*, pages 71–80, 1992.

[14] D. Dotan and R.Y. Pinter. Hyperflow: an integrated visual query and dataflow language for end-user information analysis. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005.

[15] Amineh Fadhil and Volker Haarslev. GLOO: A Graphical Query Language for OWL ontologies. *Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions*, 2006.

[16] Amineh Fadhil and Volker Haarslev. OntoVql: A Graphical Query Language for OWL Ontologies. *Proceedings of the 2007 International Workshop on Description Logics*, 810:267–274, 2007.

[17] M. Fowler, editor. *Pattern of Enterprise Application Architecture*. Pearson Education, 2003.

[18] V. Haarslev, R. Moller, and M. Wessel. Querying the semantic web with Racer + nRQL. In *CEUR Workshop Proceedings of KI-2004 Workshop on Applications of Description Logics (ADL 04)*, September 2004.

[19] http://www.jgraph.com/jgraph.html, last visited in September 2008.

[20] C.J. Kacmar and J.M. Carey. Assessing the usability of icons in user interfaces. *Behaviour and Information Technology*, 10(6):443–457, 1991.

[21] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. Rql: A declarative query language for RDF. In *The Eleventh International World Wide Web Conference (WWW)*, May 2002.

[22] E. Keramopoulos, P. Pouyioutas, and C. Sadler. GOQL, a graphical query language for object-oriented database systems. In *Basque International Workshop on Information Technology*, pages 35–45, 1997.

[23] Michael Kifer and Georg Lausen. F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 134–146, June 1989.

[24] S. Krivov and F. Villa. Towards an ontology based visual query system. *Data Integration in the Life Sciences*, pages 313–316, 2005.

[25] S. Krivov, F. Villa, and R. Williams. GrOWL, visual browser and editor for OWL ontologies. *Journal of Web Semantics*, 2006.

[26] I. Larkin and H. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 1:65–99, 1987.

[27] B. Ludascher, A. Gupta, and M.E. Martone. Model-based mediation with domain maps. *IEEE Computer Society*, 2001.

[28] W.K. Michener, J.H. Beach, M.B. Jones B. Ludaescher, D.D. Pennington, R.S. Pereira, A. Rajasekar, and M. Schildhauer. A knowledge environment for the biodiversity and ecological sciences. *Journal of Intelligent Information Systems*, pages 111–126, 2007.

[29] N. Murray, N.W. paton, C.A. Goble, and J. Bryce. Kaleidoquery: a flow-based visual language and its evaluation. *Journal of Visual Languages and Computing*, 11:151–189, 2000.

[30] Fernando Naufel and Carlos Bazilio Martins. Visualization of Description Logic Models. *Description Logics 2008*, May 2008.

[31] W. Ni and T.W. Ling. Glass: A graphical query language for semi-structured data. In *Proceedings of Eighth International Conference on Database Systems for Advanced Applications (DASFAA '03)*, March 2003.

[32] http://www.w3.org/TR/owl-guide/ , last visited in September 2008.

[33] A. Papantonakis and P.J.H. King. Gql, a declarative graphical query language based on the functional data model. In *Proc. of the Workshop on Advanced Visual Interfaces*, pages 113–122, 1994.

[34] P. R. Smart, A. Russell, D. Braines, Y. Kalfoglou, J. Bao, and N. Shadbolt. A Visual Approach to Semantic Query Design Using a Web-Based Graphical Query Designer. *16th International Conference on Knowledge Engineering and Knowledge Management*, 2008.

[35] P. R. Smart, A. Russell, D. Braines, and N. Shadbolt. NITELIGHT: A Graphical Tool for Semantic Query Construction. *Semantic Web User Interaction Workshop*, April 2008.

[36] R. Stevens, P. Baker, S. Bechhofer, G. Ng, A. Jacoby, N. Paton, C. Goble, and A. Brass. Tambis: Transparent access to multiple bioinformatics information sources. *Bioinformatics*, 16:184–189, 2000.

[37] http://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html, last visited in September 2008.

[38] M. Wessel and R. Moller. A high performance semantic web query answering engine. In *Proceedings of the 2005 International Workshop on Description Logics*, volume 147, pages 701–705, 2005.

[39] http://www.docjar.com/docs/api/java/beans/XMLDecoder.html, last visited in September 2008.

[40] http://ecoinformatics.uvm.edu/dmaps/growl/ , last visited in September 2008.

Appendix A

People-pets ontology

A.1 Source

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<rdf:RDF xmlns:ns0="http://cohse.semanticweb.org/ontologies/people#"
 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#"
 xml:base="http://cohse.semanticweb.org/ontologies/people"
 xmlns="http://cohse.semanticweb.org/ontologies/people#">
  <owl:Ontology rdf:about="" />
<owl:Class rdf:about="#white_van_man">
  <rdfs:label>white van man</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#man" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#drives" />
<owl:someValuesFrom>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#white_thing" />
  <owl:Class rdf:about="#van" />
  </owl:intersectionOf>
  </owl:Class>
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#publication">
  <rdfs:label>publication</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
  </owl:Class>
<owl:Class rdf:about="#giraffe">
  <rdfs:label>giraffe</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#animal" />
  </rdfs:subClassOf>
<rdfs:subClassOf>
<owl:Restriction>
  <owl:onProperty rdf:resource="#eats" />
<owl:allValuesFrom>
  <owl:Class rdf:about="#leaf" />
  </owl:allValuesFrom>
  </owl:Restriction>
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#cat_liker">
  <rdfs:label>cat liker</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#likes" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#cat" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#cat_owner">
  <rdfs:label>cat owner</rdfs:label>
<rdfs:comment>
```

```
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#has_pet" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#cat" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#grownup">
  <rdfs:label>grownup</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
  <owl:Class rdf:about="#adult" />
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#quality_broadsheet">
  <rdfs:label>quality broadsheet</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#broadsheet" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#vehicle">
  <rdfs:label>vehicle</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
  </owl:Class>
<owl:Class rdf:about="#newspaper">
  <rdfs:label>newspaper</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
```

```
<rdfs:subClassOf>
  <owl:Class rdf:about="#publication" />
  </rdfs:subClassOf>
<rdfs:subClassOf>
<owl:Class>
<owl:unionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#broadsheet" />
  <owl:Class rdf:about="#tabloid" />
  </owl:unionOf>
  </owl:Class>
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#bus_company">
  <rdfs:label>bus company</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#company" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#pet_owner">
  <rdfs:label>pet owner</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#has_pet" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#animal" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#mad_cow">
  <rdfs:label>mad cow</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
```

```
  <owl:Class rdf:about="#cow" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#eats" />
<owl:someValuesFrom>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#brain" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#part_of" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#sheep" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#bus">
  <rdfs:label>bus</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#vehicle" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#car">
  <rdfs:label>car</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#vehicle" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#cat">
  <rdfs:label>cat</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#animal" />
  </rdfs:subClassOf>
  </owl:Class>
```

```
<owl:Class rdf:about="#cow">
  <rdfs:label>cow</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#vegetarian" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#dog">
  <rdfs:label>dog</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
<owl:Restriction>
  <owl:onProperty rdf:resource="#eats" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#bone" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#kid">
  <rdfs:label>kid</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
  <owl:Class rdf:about="#young" />
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#man">
  <rdfs:label>man</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
  <owl:Class rdf:about="#male" />
  <owl:Class rdf:about="#adult" />
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#pet">
  <rdfs:label>pet</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Restriction>
  <owl:onProperty rdf:resource="#is_pet_of" />
<owl:someValuesFrom>
  <owl:Class rdf:about="http://www.w3.org/2002/07/owl#Thing" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#van">
  <rdfs:label>van</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#vehicle" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#company">
  <rdfs:label>company</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
  </owl:Class>
<owl:Class rdf:about="#red_top">
  <rdfs:label>red top</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#tabloid" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#bone">
  <rdfs:label>bone</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
  </owl:Class>
<owl:Class rdf:about="#duck">
  <rdfs:label>duck</rdfs:label>
<rdfs:comment>
```

```
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#animal" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#leaf">
  <rdfs:label>leaf</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
<owl:Restriction>
  <owl:onProperty rdf:resource="#part_of" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#tree" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#male">
  <rdfs:label>male</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
  </owl:Class>
<owl:Class rdf:about="#tree">
  <rdfs:label>tree</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#plant" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#broadsheet">
  <rdfs:label>broadsheet</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#newspaper" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#haulage_worker">
  <rdfs:label>haulage worker</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Restriction>
```

```
  <owl:onProperty rdf:resource="#works_for" />
<owl:someValuesFrom>
<owl:Class>
<owl:unionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#haulage_company" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#part_of" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#haulage_company" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:unionOf>
  </owl:Class>
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#haulage_truck_driver">
  <rdfs:label>haulage truck driver</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#drives" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#truck" />
  </owl:someValuesFrom>
  </owl:Restriction>
<owl:Restriction>
  <owl:onProperty rdf:resource="#works_for" />
<owl:someValuesFrom>
<owl:Restriction>
  <owl:onProperty rdf:resource="#part_of" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#haulage_company" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#bus_driver">
  <rdfs:label>bus driver</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#drives" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#bus" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#vegetarian">
  <rdfs:label>vegetarian</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#animal" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#eats" />
<owl:allValuesFrom>
<owl:Class>
<owl:complementOf>
  <owl:Class rdf:about="#animal" />
  </owl:complementOf>
  </owl:Class>
  </owl:allValuesFrom>
  </owl:Restriction>
<owl:Restriction>
  <owl:onProperty rdf:resource="#eats" />
<owl:allValuesFrom>
<owl:Class>
<owl:complementOf>
<owl:Restriction>
  <owl:onProperty rdf:resource="#part_of" />
<owl:someValuesFrom>
```

```
  <owl:Class rdf:about="#animal" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:complementOf>
  </owl:Class>
  </owl:allValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#animal_lover">
  <rdfs:label>animal lover</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#has_pet" />
  <owl:minCardinality
  rdf:datatype="http://www.w3.org/2001/XMLSchema#nonNegativeInteger">3
  </owl:minCardinality>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#dog_liker">
  <rdfs:label>dog liker</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#likes" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#dog" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#dog_owner">
  <rdfs:label>dog owner</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#has_pet" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#dog" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#elderly">
  <rdfs:label>elderly</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#adult" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#tabloid">
  <rdfs:label>tabloid</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#newspaper" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#lorry_driver">
  <rdfs:label>lorry driver</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
<owl:Restriction>
```

```
  <owl:onProperty rdf:resource="#drives" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#lorry" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#animal">
  <rdfs:label>animal</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
<owl:Restriction>
  <owl:onProperty rdf:resource="#eats" />
<owl:someValuesFrom>
  <owl:Class rdf:about="http://www.w3.org/2002/07/owl#Thing" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#driver">
  <rdfs:label>driver</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#drives" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#vehicle" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#female">
  <rdfs:label>female</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
  </owl:Class>
```

```
<owl:Class rdf:about="#adult">
  <rdfs:label>adult</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
  </owl:Class>
<owl:Class rdf:about="#brain">
  <rdfs:label>brain</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
  </owl:Class>
<owl:Class rdf:about="#grass">
  <rdfs:label>grass</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#plant" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#lorry">
  <rdfs:label>lorry</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#vehicle" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#plant">
  <rdfs:label>plant</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
  </owl:Class>
<owl:Class rdf:about="#sheep">
  <rdfs:label>sheep</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#animal" />
  </rdfs:subClassOf>
<rdfs:subClassOf>
<owl:Restriction>
  <owl:onProperty rdf:resource="#eats" />
<owl:allValuesFrom>
  <owl:Class rdf:about="#grass" />
  </owl:allValuesFrom>
  </owl:Restriction>
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#tiger">
  <rdfs:label>tiger</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#animal" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#truck">
  <rdfs:label>truck</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#vehicle" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#woman">
  <rdfs:label>woman</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
  <owl:Class rdf:about="#female" />
  <owl:Class rdf:about="#adult" />
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#young">
  <rdfs:label>young</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
  </owl:Class>
<owl:Class rdf:about="#haulage_company">
  <rdfs:label>haulage company</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#company" />
  </rdfs:subClassOf>
  </owl:Class>
```

```
<owl:Class rdf:about="#white_thing">
  <rdfs:label>white thing</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
  </owl:Class>
<owl:Class rdf:about="#person">
  <rdfs:label>person</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#animal" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#magazine">
  <rdfs:label>magazine</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#publication" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#old_lady">
  <rdfs:label>old lady</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
  <owl:Class rdf:about="#female" />
  <owl:Class rdf:about="#elderly" />
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#van_driver">
  <rdfs:label>van driver</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<owl:equivalentClass>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#person" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#drives" />
```

```
<owl:someValuesFrom>
  <owl:Class rdf:about="#van" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </owl:equivalentClass>
  </owl:Class>
<owl:Class rdf:about="#bicycle">
  <rdfs:label>bicycle</rdfs:label>
<rdfs:comment>
  </rdfs:comment>
<rdfs:subClassOf>
  <owl:Class rdf:about="#vehicle" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:ObjectProperty rdf:about="#has_child">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>has_child</rdfs:label>
  </owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#has_pet">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>has_pet</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#likes" />
<rdfs:domain>
  <owl:Class rdf:about="#person" />
  </rdfs:domain>
<rdfs:range>
  <owl:Class rdf:about="#animal" />
  </rdfs:range>
  </owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#eats">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>eats</rdfs:label>
  <owl:inverseOf rdf:resource="#eaten_by" />
<rdfs:domain>
  <owl:Class rdf:about="#animal" />
  </rdfs:domain>
  </owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#works_for">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>works_for</rdfs:label>
  </owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#has_father">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>has_father</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#has_parent" />
<rdfs:range>
  <owl:Class rdf:about="#man" />
  </rdfs:range>
  </owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#has_mother">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>has_mother</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#has_parent" />
<rdfs:range>
  <owl:Class rdf:about="#woman" />
  </rdfs:range>
  </owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#has_parent">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>has_parent</rdfs:label>
  </owl:ObjectProperty>
<owl:DatatypeProperty rdf:about="#service_number">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>service_number</rdfs:label>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#integer" />
  </owl:DatatypeProperty>
<owl:ObjectProperty rdf:about="#eaten_by">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>eaten_by</rdfs:label>
  </owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#drives">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>drives</rdfs:label>
  </owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#likes">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>likes</rdfs:label>
  </owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#reads">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>reads</rdfs:label>
<rdfs:range>
  <owl:Class rdf:about="#publication" />
  </rdfs:range>
  </owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#part_of">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>part_of</rdfs:label>
  </owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#has_part">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>has_part</rdfs:label>
  <owl:inverseOf rdf:resource="#part_of" />
  </owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#is_pet_of">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>is_pet_of</rdfs:label>
  <owl:inverseOf rdf:resource="#has_pet" />
  </owl:ObjectProperty>
<owl:Class rdf:about="#white_van_man">
<rdfs:subClassOf>
<owl:Restriction>
  <owl:onProperty rdf:resource="#reads" />
<owl:allValuesFrom>
  <owl:Class rdf:about="#tabloid" />
  </owl:allValuesFrom>
  </owl:Restriction>
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#driver">
<rdfs:subClassOf>
  <owl:Class rdf:about="#adult" />
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#old_lady">
<rdfs:subClassOf>
<owl:Class>
<owl:intersectionOf rdf:parseType="Collection">
```

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#has_pet" />
<owl:allValuesFrom>
  <owl:Class rdf:about="#cat" />
  </owl:allValuesFrom>
  </owl:Restriction>
<owl:Restriction>
  <owl:onProperty rdf:resource="#has_pet" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#animal" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:intersectionOf>
  </owl:Class>
  </rdfs:subClassOf>
  </owl:Class>
<owl:Class rdf:about="#dog">
<owl:disjointWith>
  <owl:Class rdf:about="#cat" />
  </owl:disjointWith>
  </owl:Class>
<owl:Class rdf:about="#broadsheet">
<owl:disjointWith>
  <owl:Class rdf:about="#tabloid" />
  </owl:disjointWith>
  </owl:Class>
<owl:Class>
<owl:unionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#animal" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#part_of" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#animal" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:unionOf>
<owl:disjointWith>
<owl:Class>
<owl:unionOf rdf:parseType="Collection">
  <owl:Class rdf:about="#plant" />
<owl:Restriction>
  <owl:onProperty rdf:resource="#part_of" />
<owl:someValuesFrom>
  <owl:Class rdf:about="#plant" />
  </owl:someValuesFrom>
  </owl:Restriction>
  </owl:unionOf>
  </owl:Class>
  </owl:disjointWith>
  </owl:Class>
<owl:Class rdf:about="#adult">
<owl:disjointWith>
  <owl:Class rdf:about="#young" />
  </owl:disjointWith>
  </owl:Class>
<rdf:Description rdf:about="#The_Times">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>The Times</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#broadsheet" />
  </rdf:type>
  </rdf:Description>
<rdf:Description rdf:about="#The_Sun">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>The Sun</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#tabloid" />
  </rdf:type>
  </rdf:Description>
<owl:Thing rdf:about="#Daily_Mirror">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Daily Mirror</rdfs:label>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Thing" />
  </owl:Thing>
<rdf:Description rdf:about="#Q123_ABC">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Q123 ABC</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#van" />
  </rdf:type>
<rdf:type>
  <owl:Class rdf:about="#white_thing" />
  </rdf:type>
  </rdf:Description>
<rdf:Description rdf:about="#Joe">
<rdfs:comment>
```

```
  </rdfs:comment>
  <rdfs:label>Joe</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#person" />
  </rdf:type>
<rdf:type>
<owl:Restriction>
  <owl:onProperty rdf:resource="#has_pet" />
  <owl:maxCardinality
  rdf:datatype="http://www.w3.org/2001/XMLSchema#nonNegativeInteger">1
  </owl:maxCardinality>
  </owl:Restriction>
  </rdf:type>
  <ns0:has_pet rdf:resource="#Fido" />
  </rdf:Description>
<rdf:Description rdf:about="#Rex">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Rex</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#dog" />
  </rdf:type>
  <ns0:is_pet_of rdf:resource="#Mick" />
  </rdf:Description>
<owl:Thing rdf:about="#Tom">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Tom</rdfs:label>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Thing" />
  </owl:Thing>
<rdf:Description rdf:about="#Flossie">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Flossie</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#cow" />
  </rdf:type>
  </rdf:Description>
<rdf:Description rdf:about="#Fido">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Fido</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#dog" />
  </rdf:type>
  </rdf:Description>
<rdf:Description rdf:about="#Fred">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Fred</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#person" />
  </rdf:type>
  <ns0:has_pet rdf:resource="#Tibbs" />
  </rdf:Description>
<rdf:Description rdf:about="#Huey">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Huey</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#duck" />
  </rdf:type>
  </rdf:Description>
<rdf:Description rdf:about="#Mick">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Mick</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#male" />
  </rdf:type>
  <ns0:drives rdf:resource="#Q123_ABC" />
  <ns0:reads rdf:resource="#Daily_Mirror" />
  </rdf:Description>
<rdf:Description rdf:about="#Walt">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Walt</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#person" />
  </rdf:type>
  <ns0:has_pet rdf:resource="#Huey" />
  <ns0:has_pet rdf:resource="#Dewey" />
  <ns0:has_pet rdf:resource="#Louie" />
  </rdf:Description>
<rdf:Description rdf:about="#The_Guardian">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>The Guardian</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#broadsheet" />
  </rdf:type>
  </rdf:Description>
<rdf:Description rdf:about="#Fluffy">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Fluffy</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#tiger" />
  </rdf:type>
  </rdf:Description>
<rdf:Description rdf:about="#Minnie">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Minnie</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#elderly" />
  </rdf:type>
<rdf:type>
  <owl:Class rdf:about="#female" />
  </rdf:type>
  <ns0:has_pet rdf:resource="#Tom" />
  </rdf:Description>
<rdf:Description rdf:about="#Dewey">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Dewey</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#duck" />
  </rdf:type>
  </rdf:Description>
<rdf:Description rdf:about="#Kevin">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Kevin</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#person" />
  </rdf:type>
  </rdf:Description>
<rdf:Description rdf:about="#Louie">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Louie</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#duck" />
  </rdf:type>
  </rdf:Description>
<rdf:Description rdf:about="#The42">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>The42</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#bus" />
  </rdf:type>
  <ns0:service_number
  rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">42
  </ns0:service_number>
  </rdf:Description>
<rdf:Description rdf:about="#Tibbs">
<rdfs:comment>
  </rdfs:comment>
  <rdfs:label>Tibbs</rdfs:label>
<rdf:type>
  <owl:Class rdf:about="#cat" />
  </rdf:type>
  </rdf:Description>
  </rdf:RDF>
```

A.2 TBox and ABox

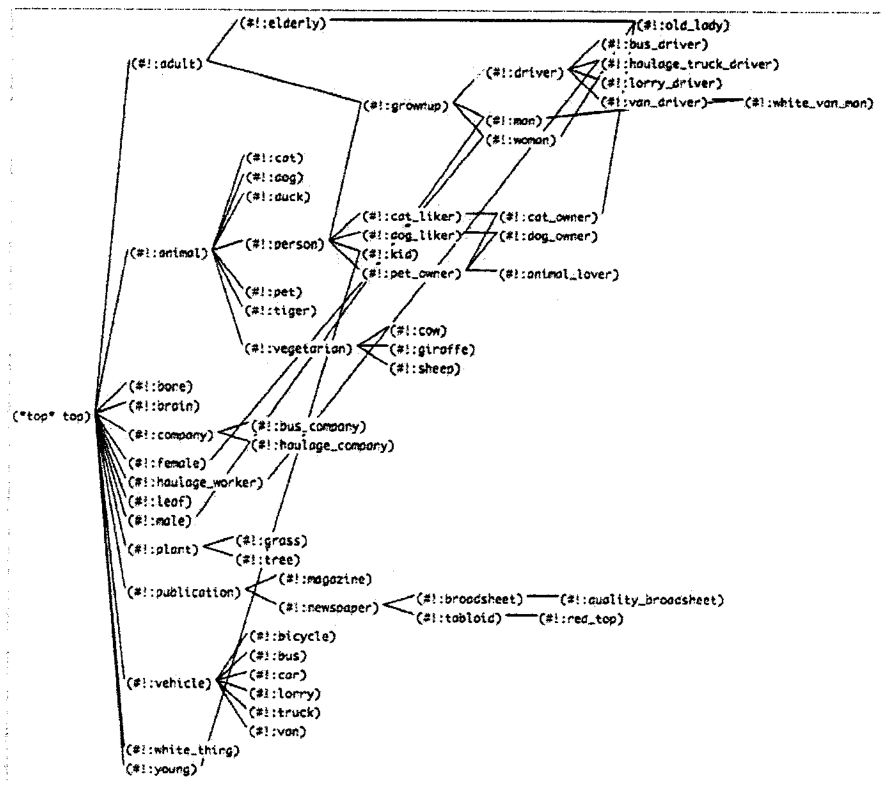
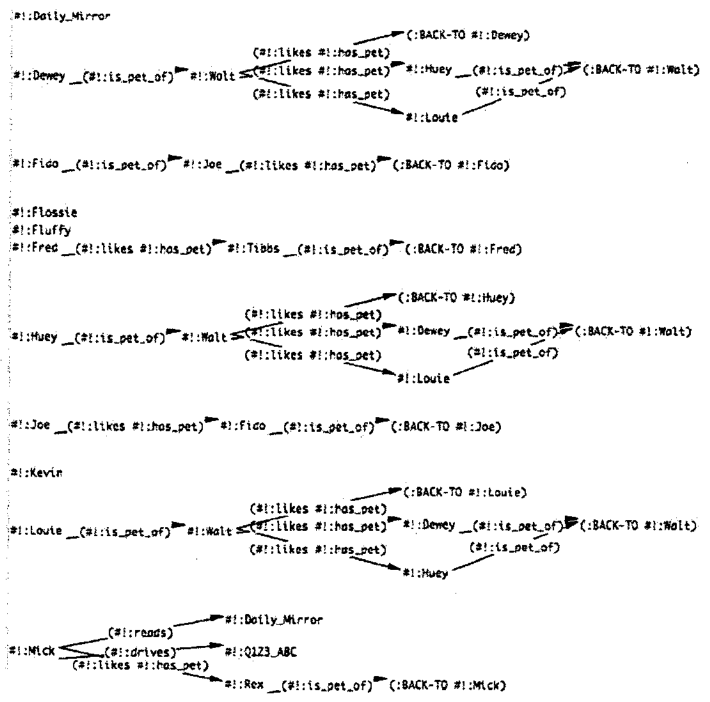
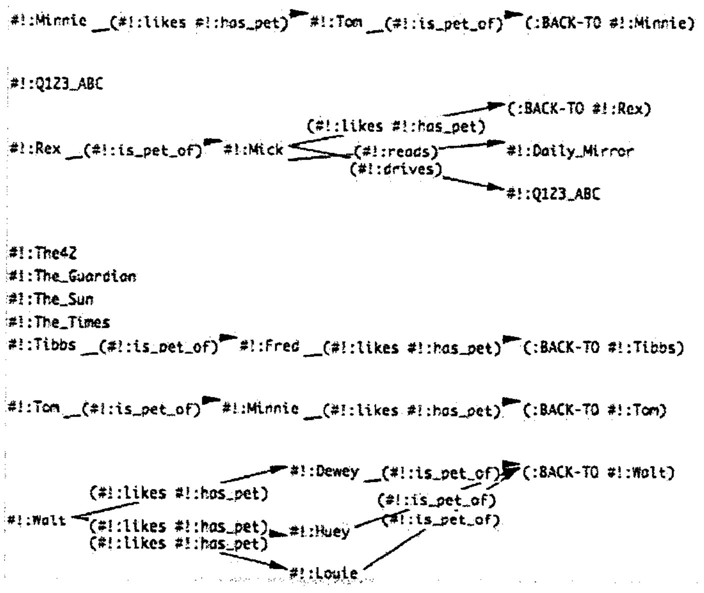


Figure A.1: TBox graph of the People-pets.owl ontology


#!:Daily_Mirror
(:BACK-TO #!:Dewey)
(#!:likes #!:has_pet)
#!:Dewey __(#!:is_pet_of) #!:Walt (#!:likes #!:has_pet) #!:Huey __(#!:is_pet_of) (:BACK-TO #!:Walt)
(#!:likes #!:has_pet)
(#!:is_pet_of)
#!:Louie
#!:Fido __(#!:is_pet_of) #!:Joe __(#!:likes #!:has_pet) (:BACK-TO #!:Fido)
#!:Flossie
#!:Fluffy
#!:Fred __(#!:likes #!:has_pet) #!:Tibbs __(#!:is_pet_of) (:BACK-TO #!:Fred)
(:BACK-TO #!:Huey)
(#!:likes #!:has_pet)
#!:Huey __(#!:is_pet_of) #!:Walt (#!:likes #!:has_pet) #!:Dewey __(#!:is_pet_of) (:BACK-TO #!:Walt)
(#!:likes #!:has_pet)
(#!:is_pet_of)
#!:Louie
#!:Joe __(#!:likes #!:has_pet) #!:Fido __(#!:is_pet_of) (:BACK-TO #!:Joe)
#!:Kevin
(:BACK-TO #!:Louie)
(#!:likes #!:has_pet)
#!:Louie __(#!:is_pet_of) #!:Walt (#!:likes #!:has_pet) #!:Dewey __(#!:is_pet_of) (:BACK-TO #!:Walt)
(#!:likes #!:has_pet)
(#!:is_pet_of)
#!:Huey
#!:Daily_Mirror
(#!:reads)
#!:Mick (#!:drives) #!:Q123_ABC
(#!:likes #!:has_pet)
#!:Rex __(#!:is_pet_of) (:BACK-TO #!:Mick)

(a)


#!:Minnie __(#!:likes #!:has_pet) #!:Tom __(#!:is_pet_of) (:BACK-TO #!:Minnie)
#!:Q123_ABC
(:BACK-TO #!:Rex)
(#!:likes #!:has_pet)
#!:Rex __(#!:is_pet_of) #!:Mick (#!:reads) #!:Daily_Mirror
(#!:drives)
#!:Q123_ABC
#!:The42
#!:The_Guardian
#!:The_Sun
#!:The_Times
#!:Tibbs __(#!:is_pet_of) #!:Fred __(#!:likes #!:has_pet) (:BACK-TO #!:Tibbs)
#!:Tom __(#!:is_pet_of) #!:Minnie __(#!:likes #!:has_pet) (:BACK-TO #!:Tom)
#!:Dewey __(#!:is_pet_of) (:BACK-TO #!:Walt)
(#!:likes #!:has_pet)
(#!:is_pet_of)
#!:Walt (#!:likes #!:has_pet) #!:Huey (#!:is_pet_of)
(#!:likes #!:has_pet)
#!:Louie

(b)

Figure A.2: ABox graphs of the People-pets.owl ontology